Calcul du n-ème entier de Hamming

Michel Quercia
quercia@cal.enst.fr
Juin 1999

Chapitre 1

Introduction

Le problème de Hamming est le suivant : *Un entier de Hamming est un entier naturel dont les seuls diviseurs premiers sont* 2, 3 *et* 5. *Écrire un programme imprimant dans l'ordre les n premiers entiers de Hamming.*

Ce problème est présenté comme un exercice de programmation et comme une illustration du calcul paresseux. Il a été introduit par Dijkstra (*A discipline of programming*, Prentice Hall 1976, pp. 129-134). L'algorithme naïf qui consiste à passer en revue tous les nombres entiers dans l'ordre en ne retenant que ceux de la forme $p = 2^x 3^y 5^z$ (ce qui peut être testé en divisant p par 2, 3, 5 autant de fois que possible et en vérifiant que le dernier quotient vaut 1) n'est pas raisonnable : le 2000-ème entier de Hamming est égal à 8 100 000 000, le 2001-ème à 8 153 726 976 c'est à dire qu'il faudrait tester plus de 53 millions de nombres avant d'augmenter la liste des entiers de Hamming d'un élément. Les algorithmes que l'on trouve sur Internet à ce sujet reposent sur l'équation :

$$\mathcal{H} = \{1\} \cup 2\mathcal{H} \cup 3\mathcal{H} \cup 5\mathcal{H}$$

où $\mathcal{H}$ désigne l'ensemble des entiers de Hamming. Cette équation doit être interprétée comme suit : $\mathcal{H}$ est le plus petit ensemble contenant 1 et stable par multiplication par 2, 3 ou 5 (on considère ici que 1 est un entier de Hamming ayant le rang zéro). En particulier le n-ème entier de Hamming est 1 si $n = 0$, sinon c'est le plus petit entier double ou triple ou quintuple d'un entier de Hamming de rang inférieur ou égal à $n-1$. Pour calculer le n-ème entier de Hamming il suffit donc de calculer les $n-1$ précédents, de les multiplier par 2, 3 et par 5 et de retenir le plus petit entier non encore vu parmi ces produits. On verra ci-dessous que cet algorithme peut être implémenté avec une complexité $O(n)$ en temps d'exécution et $O(n^{2/3})$ en mémoire (l'unité de temps est une opération entre entiers de Hamming et l'unité de mémoire est un entier de Hamming).

Une variante du problème de Hamming consiste à calculer le n-ème entier de Hamming le plus rapidement possible sans calculer nécessairement tous les précédents. Une interprétation géométrique des entiers de Hamming comme points à coordonnées entières de l'espace tridimensionnel classés selon la forme linéaire $(x, y, z) \mapsto x \ln 2 + y \ln 3 + z \ln 5$ permet alors de déterminer efficacement une plage plus ou moins restreinte dans laquelle se trouve le n-ème point de Hamming et d'identifier ce point par recherche dans la plage déterminée ce qui conduit à un algorithme de recherche ayant une complexité $O(n^{1/3} \ln n)$ en temps et $O(\ln n)$ en mémoire.

Ces deux problèmes de Hamming ont donné lieu à une discussion passionnée sur la liste de diffusion `ups-info` en 1998 avec présentation d'algorithmes de plus en plus performants et l'objet de ce rapport est de rendre compte de ces algorithmes. Les mesures de temps d'exécution et d'occupation mémoire données ci-après ont été faites un PC-Linux (Pentium 233-MMX, 64Mo de Ram) à partir d'une implémentation Ocaml de chaque algorithme compilée avec `ocamlopt` version 2.02. Les programmes sont listés intégralement au chapitre 5, et sont disponibles sur le Web à l'URL `http://pauillac.inria.fr/~quercia/hamming.tgz`.

Chapitre 2

Calcul des n premiers entiers de Hamming

2.1 Calcul paresseux

Le principe du calcul paresseux consiste à différer un calcul jusqu'au moment où son résultat devient nécessaire pour continuer. Une fois le calcul effectué le résultat est conservé pour utilisation ultérieure sans re-calcul. En Ocaml, la bibliothèque `Lazy` implémente les valeurs paresseuses. On crée une telle valeur par l'expression :

```
let x = lazy(expr)
```

où `expr` est une expression Ocaml quelconque. Cette expression n'est pas évaluée immédiatement, mais les instructions permettant son évaluation sont enregistrées dans `x`. On obtient la valeur de `expr` par l'expression :

```
let y = force(x)
```

A ce moment `expr` est évaluée et `y` est lié au résultat. Ce résultat est aussi conservé dans `x` et les appels ultérieurs `force(x)` le retourneront sans provoquer de re-calcul.

Le calcul paresseux permet d'implémenter des structures de données potentiellement infinies dont on connaît le début et le moyen de calculer la suite. Par exemple la liste $\mathcal{H}$ des entiers de Hamming est définie paresseusement comme ayant une tête égale à 1 et une queue égale à la fusion sans répétition de $2\mathcal{H}$, $3\mathcal{H}$ et $5\mathcal{H}$. Le programme `listp` page 25, dû à Pierre Weis, est d'une étonnante simplicité.

exécution :

```
bash$ time -f "temps=%es" ./listp 1000000
519381797917090766274082018159448243742493816603938969000000000000000000000000000000
temps=81.88s
```

Validité et complexité : On note 2`h`, 3`h` et 5`h` les listes `mult (Int 2) h` `mult (Int 3) h`, et `mult (Int 5) h`. A un instant donné on suppose qu'on a déjà calculé les n premiers éléments de `h` et que l'on a extrait a éléments de 2`h`, b éléments de 3`h` et c éléments de 5`h` avec $a < n$, $b < n$ et $c < n$. Cette hypothèse est vérifiée au moment où l'on construit `h` avec $n = 1$, $a = b = c = 0$.

Lors du calcul du n-ème élément de `h` (élément de rang $n+1$) on dispose déjà des éléments de rangs $a+1$ de 2`h`, $b+1$ de 3`h` et $c+1$ de 5`h` ou on les obtient par au plus trois multiplications car $a < n$, $b < n$ et $c < n$ donc le n-ème élément de `h` s'obtient en temps borné (deux comparaisons et au plus trois multiplications). Ceci prouve déjà que l'appel `nème n h` termine au bout d'un temps $O(n)$.

Ensuite, on prouve aisément par récurrence que tous les éléments de `h` sont des entiers de Hamming et qu'ils forment une suite strictement croissante. Enfin, si x est un élément de cette suite alors $2x$ figure dans `h` sinon tous les éléments de 3`h` et 5`h` seraient inférieurs à x ce qui contredit la propriété de croissance stricte. De même $3x$ et $5x$ figurent aussi dans `h` donc `h` est stable par multiplication par 2, 3 et 5 et contient 1 comme premier élément, c'est bien la suite de Hamming.

Complexité spatiale : à priori la complexité spatiale est aussi $O(n)$ puisque chaque étape consome une quantité bornée de mémoire. En fait, la liste `h` étant locale au calcul du n-ème entier de Hamming, la mémoire occupée par le début de cette liste est récupérable par le gestionnaire de mémoire intégré à Ocaml au fur et à mesure que l'on progresse dans les listes 2`h`, 3`h` et 5`h`. Donc si x est le n-ème entier de Hamming seuls les entiers de Hamming compris entre $x/5$ et $x-1$ doivent être conservés pour permettre le calcul de x et des nombres suivants. On verra dans la partie 3.1 que le nombre de tels entiers est de l'ordre de $n^{2/3}$ c'est à dire que le programme de P. Weis *peut tourner* avec $O(n^{2/3})$ unités de mémoire. Expérimentalement une augmentation de n dans un rapport 8 produit une augmentation de la mémoire allouée dans un rapport de l'ordre de 4 (cf. tableau ci-dessous). Par contre, si `h` est déclarée au niveau global alors tous les termes calculés sont conservés et la mémoire consommée augmente linéairement avec n.

	40	80	160	320	640	1 280	2 560	n (milliers)
listp	2.24	4.92	10.66	23.35	49.49	103.79	218.35	temps (s)
	1 412	1 676	1 936	2 964	4 500	7 296	11 600	mémoire (Ko)
listp1	2.90	6.41	13.58	31.11	101.17			temps (s)
	4 828	9 720	18 640	39 356	91 512			mémoire (Ko)

```
(* fichier listp1.ml : identique à listp.ml sauf la liste h qui est ici globale *)
let rec h =
  {hd = Int 1;
   tl = lazy((mult (Int 2) h) @ (mult (Int 3) h) @ (mult (Int 5) h))}
let _ =
  let n = int_of_string(Sys.argv.(1)) in
  print_string(string_of_num(nème n h));
  print_newline()
```

2.2 Files de priorité

Le programme de Pierre Weis a été le point de départ d'une compétition entre différents abonnés de la liste `ups-info` pour calculer le plus rapidement possible les n premiers nombres de Hamming. Les deux programmes qui suivent reposent sur les mêmes principes :

- Le nombre de Hamming $x = 2^a 3^b 5^c$ est représenté en mémoire par les trois entiers a, b et c et son logarithme népérien noté ℓ. Multiplier x par 2, 3 ou 5 revient à ajouter 1 à l'exposant adéquat et $\ln 2$, $\ln 3$ ou $\ln 5$ à ℓ. La comparaison de deux entiers de Hamming se fait sur leurs logarithmes.
- On tient à jour la liste `h` des entiers de Hamming double, triple ou quintuple d'un entier de Hamming déjà vu. Pour produire un nouvel entier on retire de `h` son plus petit élément, x, et on y insère ses multiples $2x$, $3x$ et $5x$. La liste initiale est réduite à 1.
- Pour éviter d'insérer plusieurs fois le même nombre on n'insère $2x$ que si x est une puissance de 2, $3x$ que si x est le produit d'une puissance de 2 par une puissance de 3 et $5x$ dans tous les cas.

Il est clair qu'on ne produit ainsi que des entiers de Hamming et qu'on les produit par ordre croissant. On prouve qu'on les produit tous jusqu'au rang n par récurrence sur n : pour $n = 0$ il n'y a rien à prouver. Si l'hypothèse est vraie au rang n alors après extraction du n-ème élément, x, `h` contient tous les quintuples des entiers de Hamming compris entre $x + 1$ et $5x$ et certains doubles et triples eux-aussi compris entre $x + 1$ et $5x$. Soit y le premier entier de Hamming suivant x ; il s'agit de prouver que y est dans `h`.

- Si y est divisible par 5, $y = 5t$ et $t \leqslant x$ (car $5x$ est un entier de Hamming suivant x et y est le plus petit) donc y a été inséré dans `h` et il y est encore puisqu'on n'a retiré que des nombres inférieurs ou égaux à x.
- Si y n'est pas divisible par 5 mais l'est par 3, $y = 3t$ avec $t \leqslant x$ et t n'est pas divisible par 5 donc on a inséré $3t$ dans `h` lors de l'extraction de t.
- Sinon y est une puissance de 2, $y = 2t$ où t est lui aussi une puissance de 2 donc $2t$ a été inséré dans `h` lors de l'extraction de t.

Le raisonnement précédent prouve aussi qu'un entier de Hamming n'est inséré qu'une seule fois dans `h`.

Remarque : une critique potentielle contre les deux programmes qui suivent est que les logarithmes étant calculés de façon approchée il est concevable qu'une erreur d'arrondi provoque un mauvais classement. Expérimentalement ce n'est pas le cas pour le premier million d'entiers de Hamming mais une justification théorique semble délicate. On pourrait implémenter une comparaison approchée avec contrôle de précision et recourir à l'arithmétique exacte en cas de précision insuffisante au prix d'une dégradation sévère de la vitesse de calcul.

2.2.1 Ensembles ordonnés

Une première méthode consiste à implémenter `h` par un ensemble à l'aide de la bibliothèque `Set` de Ocaml. Cette bibliothèque représente les ensembles sur un type totalement ordonné par des arbres de recherche équilibrés, donc les temps d'insertion d'un élément et d'extraction du minimum sont logarithmiques en la taille de l'ensemble. Dans le cas présent, comme `h` ne contient que des nombres de Hamming compris entre $x+1$ et $5x$ où x est le dernier élément extrait, la taille de `h` est $O(n^{2/3})$ où n est le rang du dernier nombre extrait donc la gestion de `h` a un coût temporel $O(\ln n)$ pour une opération (insertion ou extraction d'un élément) et le temps total de calcul des n premiers entiers de Hamming un coût temporel $O(n \ln n)$. La complexité spatiale est $O(n^{2/3})$. Le programme correspondant est listé page 25.

exécution :

```
bash$ time -f "temps=%es" ./prifile 1000000
2^38 . 3^109 . 5^29
temps=37.31s
```

mesures de temps et mémoire :

40	80	160	320	640	1 280	2 560	n (milliers)
1.20	2.49	5.30	11.21	22.78	47.25	98.12	temps (s)
1 084	1 308	1 588	1 840	2 600	3 860	5 624	mémoire (Ko)

2.2.2 Listes mutables

La deuxième implémentation de `h`, proposée par Denis Cazor, utilise une liste mutable (c'est à dire une liste dont les pointeurs de chaînage sont mutables de façon à pouvoir insérer un élément en milieu de liste sans recopier le début de la liste), classée par ordre croissant. De plus Denis remarque que lorsqu'on insère $5t$ dans `h` alors c'est le dernier élément de `h` donc on peut le placer directement à la fin pourvu qu'on dispose d'un pointeur, z, vers le dernier élément de `h`. Pour insérer $3t$ on peut partir de la dernière position, y, où l'on a inséré un triple $3t'$ car nécessairement $t' < t$. On progresse à partir de y jusqu'à trouver deux entiers encadrant $3t$ et on l'insère à cet endroit en mettant à jour y. L'insertion d'un double se fait de même à l'aide d'un pointeur, x, mémorisant la dernière position où l'on a inséré un double. Voir le listing page 26.

Remarques :

- Ocaml ne permettant pas d'utiliser des données non initialisées, le dernier élément de `h` doit contenir dans le champ `tl` un pointeur vers une cellule de type `listm` bien que l'on n'utilise pas cette cellule pointée. La liste cyclique `nil` est utilisée ici comme cible de ce dernier pointeur.

- La fonction `insère` n'est appelée que pour insérer un double $2t$ ou un triple $3t$, et ceci après qu'on ait placé $5t$ en queue de liste donc il est certain que l'on ne dépassera pas la fin de la liste.

exécution :

```
bash$ time -f "temps=%es" ./cazor 1000000
2^38 . 3^109 . 5^29
temps=7.51s
```

mesures de temps et mémoire :

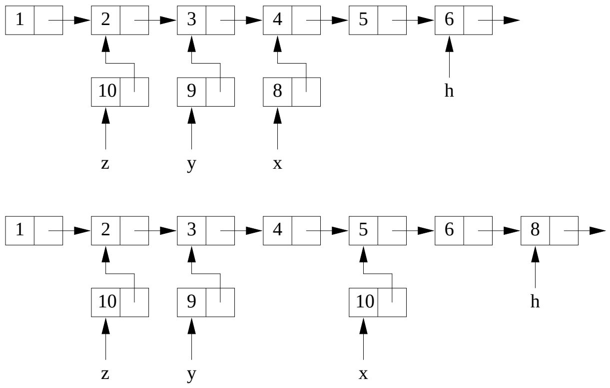
40	80	160	320	640	1 280	2 560	n (milliers)
0.22	0.44	0.92	2.01	4.35	9.10	19.03	temps (s)
1 084	1 352	1 376	1 588	2 400	3 164	4 208	mémoire (Ko)

Complexité temporelle : Les mesures précédentes laissent penser que la complexité temporelle du programme de Denis est linéaire par rapport à n. Ceci n'est pas évident au vu du code car le temps de recherche de la position d'insertion d'un double ou un triple est difficile à borner de façon linéaire : il y a $O(n^{2/3})$ entiers de Hamming inférieur au n-ème non divisibles par 5 et la liste `h` est de taille $O(n^{2/3})$ donc le temps cumulé des insertions de doubles et triples est $O(n^{4/3})$.

En fait si t est le n-ème nombre de Hamming alors les pointeurs x et y n'auront pas dépassé le nombre $5t$ et le nombre d'entiers de Hamming inférieurs à $5t$ est $O(n)$ donc le temps cumulé des insertions de doubles et triples est effectivement $O(n)$. Par ailleurs, on peut remarquer que lorsqu'on doit insérer un double, $2t$, alors t est une puissance de 2 et x pointe vers le dernier double inséré c'est à dire vers t qui est aussi pointé par h. On pourrait donc supprimer la gestion de x dans le programme, expérimentalement cela n'apporte pas d'accélération sensible.

2.3 Classement sans comparaison

L'algorithme suivant, listé page 27 et proposé par Damien Doligez, reprend la méthode de Pierre Weis : la liste $\mathcal{H}$ des entiers de Hamming est prolongée élément par élément en prenant le plus petit des entiers non encore sélectionnés dans $2\mathcal{H}$, $3\mathcal{H}$ et $5\mathcal{H}$, ces listes étant elles-mêmes déduites de $\mathcal{H}$. La première différence avec la méthode de Pierre Weis est qu'il n'y a pas de code stocké en bout de liste mais seulement un pointeur prêt à être modifié pour pointer vers les éléments ajoutés et c'est le code du programme principal qui ajoute ces éléments l'un après l'autre par une simple boucle. On économise ainsi la création et la gestion des suspensions (code à exécuter plus tard et environnement associé) d'où une charge machine moindre et une exécution plus rapide. Par rapport aux méthodes avec file de priorité on économise le calcul inutile des doubles, triples ou quintuples dépassant le n-ème entier de Hamming.



Le principe de fonctionnement est illustré par la figure ci-dessus : La liste $\mathcal{H}$ de Hamming est en cours de construction et `h` pointe vers le dernier élément calculé ; `x` pointe vers le premier élément, $2x$, de la liste $2\mathcal{H}$ non encore utilisé, cet élément pointant vers l'élément x de $\mathcal{H}$ dont il est issu ; de même pour `y`, $3y$, `z` et $5z$. On fait progresser la construction en plaçant derrière la cellule pointée par `h` le nombre $m = \min(2x, 3y, 5z)$, en faisant avancer `h` d'un cran, et en mettant à jour les cellules pointées par `x`, `y` et `z` selon que $m = 2x$, $m = 3y$ ou $m = 5z$ (ces cas ne sont pas exclusifs). Au bout de n étapes `h` pointe vers le n-ème nombre de Hamming. La démonstration de validité de cet algorithme est identique à celle pour le programme de Pierre Weis.

Une deuxième amélioration présentée par Damien est une technique de calcul de m qui ne fait quasiment jamais appel à la comparaison entre entiers de Hamming. Il remarque que si $2x$ est divisible par 3 alors $2x = 3t$ et le nombre $3t$ figure dans $3\mathcal{H}$ après $3y$ sinon il aurait déjà été placé dans la liste résultat ($3y$ est le premier triple non placé). Donc $2x = 3t \geqslant 3y$ et par conséquent $m = \min(3y, 5z)$. De même si $3y$ est un multiple de 5 alors il est supérieur ou égal à $5z$ d'où $m = 5z$ et si $5z$ est un multiple de 3 alors $m = 3y$. On arrive ainsi à déterminer m sans comparaison en examinant les exposants de 2, 3 et 5 des entiers $2x$, $3y$ et $5z$ au moins dans tous les cas où ces trois nombres ne sont pas primaires (ayant un seul facteur premier). En pratique cette technique de calcul rapide de m échoue seulement 52 fois sur le premier million d'entiers de Hamming.

exécution :

```
bash$ time -f "temps=%es" ./nocomp 1000000
2^38 . 3^109 . 5^29
temps=4.96s
```

mesures de temps et mémoire :

40	80	160	320	640	1 280	2 560	n (milliers)
0.18	0.33	0.69	1.46	3.04	6.25	13.17	temps (s)
888	908	1 140	1 392	1 644	1 896	2 625	mémoire (Ko)

Chapitre 3

Méthodes géométriques

3.1 Tétraèdres de Hamming

Un entier de Hamming, $2^x 3^y 5^z$, peut être vu comme un point de l'espace de coordonnées x, y, z. L'ensemble des entiers de Hamming inférieurs ou égaux à une borne K correspond à l'ensemble des points à coordonnées entières dans le tétraèdre limité par les plans d'équations :

$$x = 0, \qquad y = 0, \qquad z = 0, \qquad x \lg 2 + y \lg 3 + z \lg 5 \leqslant \lg K$$

où lg désigne le logarithme en base 2. Cette interprétation permet déjà d'évaluer approximativement le nombre $n(K)$ d'entiers de Hamming majorés par K à l'aide du volume de ce tétraèdre :

$$v(K) = \frac{(\lg K)^3}{6 \lg 2 \lg 3 \lg 5}.$$

Chaque point entier dans le tétraèdre est le sommet le plus proche de l'origine d'un cube de côté 1 et la réunion de ces cubes recouvre le tétraèdre donc $v(K) \leqslant n(K)$. Mais tout sommet de cette réunion de cubes est inclus dans le tétraèdre limité par le plan d'équation :

$$x \lg 2 + y \lg 3 + z \lg 5 \leqslant \lg K + \lg 2 + \lg 3 + \lg 5 = \lg(30K)$$

donc $n(K) \leqslant v(30K)$. On en déduit :

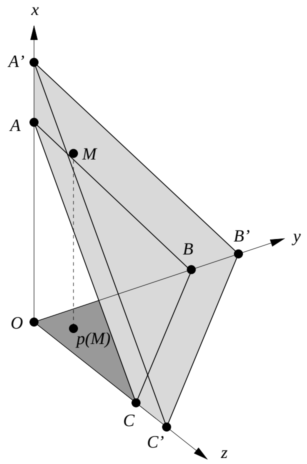
$$n(K) = \frac{(\lg K)^3}{6 \lg 2 \lg 3 \lg 5} + O((\lg K)^2)$$

et, pour $\alpha > 1$ constant :

$$n(\alpha K) - n(K) \leqslant v(30\alpha K) - v(K) = O((\lg K)^2).$$

Si t est le n-ème nombre de Hamming et $K = \lg t$ alors $n(K) = n + 1$ donc $\lg K \sim \sqrt[3]{6n \lg 2 \lg 3 \lg 5}$ ce qui prouve que le nombre d'entiers de Hamming compris entre t et αt est $O(n^{2/3})$ pour $\alpha > 1$ fixé. Les estimations de complexité des algorithmes de calcul des n premiers entiers de Hamming résultent de cette majoration.

3.2 Recherche par comptage



La figure ci-dessus illustre cette méthode. Pour déterminer le n-ème entier de Hamming on compte le nombre de points dans un tétraèdre de Hamming de paramètre $K = 2^k$ par récurrence sur k et l'on fait progresser k tant que $n(2^k) < n$. Soit k tel que $n(2^{k-1}) < n \leqslant n(2^k)$, O, A, B, C les sommets du tétraèdre de paramètre 2^{k-1} et O, A', B', C' ceux du tétraèdre de paramètre 2^k. On sait donc que le n-ème nombre de Hamming est représenté par un point entier dans la région gris-clair, il suffit de classer tous les points entiers dans cette région selon l'ordre induit par celui des entiers de Hamming et de chercher l'élément de rang $n(2^k) - n$ en partant de la fin.

Puisqu'on utilise le logarithme en base 2, A et A' ont pour abscisses $k-1$ et k, donc l'épaisseur en x de la tranche $ABCA'B'C'$ vaut 1 et tous les points entiers de cette tranche se projettent parallèlement à Ox de façon bijective sur les points entiers du triangle $OB'C'$. On en déduit la relation de récurrence :

$$n(2^k) = n(2^{k-1}) + p(2^k)$$

où $p(2^k)$ est le nombre de points (y, z) entiers vérifiant les inéquations :

$$y \geqslant 0, \qquad z \geqslant 0, \qquad y \lg 3 + z \lg 5 \leqslant k.$$

En comptant ces points selon z on a :

$$p(2^k) = \sum_{z=0}^{\lfloor k/\lg 5 \rfloor} \Bigl(1 + \Bigl\lfloor \frac{k - z \lg 5}{\lg 3} \Bigr\rfloor\Bigr).$$

En supposant les calculs en flottant suffisamment précis, $p(2^k)$ est calculable en temps $O(k)$ donc k et $n(2^k)$ sont calculables en temps $O(k^2) = O(n^{2/3})$. La tranche $ABCA'B'C'$ contient $O(n^{2/3})$ points entiers déduits de ceux du triangle $OB'C'$, ces points sont classables en temps $O(n^{2/3} \ln n)$ et la recherche du point de rang $n(2^k) - n$ se résume à un parcours dans la liste classée d'où un temps $O(n^{2/3})$. On peut donc déterminer le n-ème entier de Hamming en temps $O(n^{2/3} \ln n)$ et en mémoire $O(n^{2/3})$, le programme correspondant est listé page 28.

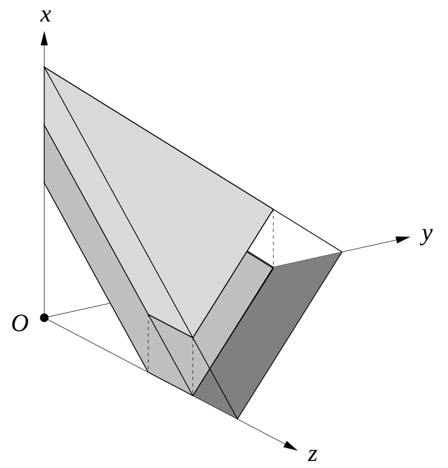
exécution :

```
bash$ time -f "temps=%es" ./comptage 1000000
2^38 . 3^109 . 5^29
temps=0.54s
```

mesures de temps et mémoire :

1	2	4	8	16	32	64	n (millions)
0.54	1.05	1.79	3.16	5.75	9.59	17.17	temps (s)
2 372	3 052	4 300	6 480	9 956	15 128	23 740	mémoire (Ko)

3.3 Comptage et classement simultanés



Damien Doligez améliore la méthode précédente en réalisant simultanément le comptage des points dans les tétraèdres successifs et le classement de la dernière tranche, le tout en arithmétique exacte ! Il remarque qu'une tranche de Hamming (points compris entre les tétraèdres de paramètres 2^{k-1} et 2^k) s'obtient à partir de la tranche précédente par translation d'une unité le long de l'axe des x et adjonction des points de la nouvelle tranche situés dans le plan yOz. En termes d'entiers de Hamming, cela signifie que l'ensemble des entiers de Hamming dans l'intervalle $]2^{k-1}, 2^k]$ est la réunion des doubles des entiers de Hamming de l'intervalle précédent $]2^{k-2}, 2^{k-1}]$ et des entiers de la forme $3^y 5^z$ compris entre 2^{k-1} et 2^k. On passe ainsi d'une tranche triée à la suivante par une translation qui ne modifie pas le classement et par insertion au bon endroit des points associés aux entiers de Hamming impairs. Ceux-cis peuvent être générés dans l'ordre par une adaptation des méthodes du chapitre précédent, il reste à déterminer s'ils appartiennent bien à la tranche en cours et à quel endroit les insérer.

Soit $h = 3^b 5^c$ un entier à insérer. On suppose $b > 0$ c'est à dire h divisible par 3. Donc $t = h/3$ est un entier de Hamming qui a été engendré avant h

(dans la tranche précédente ou dans celle d'avant) et on dispose déjà dans la tranche en cours du nombre $2^\alpha t$ avec $\alpha = 1$ ou $\alpha = 2$. Soit $2^\beta u$ l'entier de Hamming précédant $2^\alpha t$ avec u impair. On a déjà inséré $3u$ dans une tranche antérieure et on dispose donc de $v = 2^\gamma.3u$ dans la tranche courante. Enfin on a $2^\beta u < 2^\alpha t = 2^\alpha h/3$ donc $2^{\beta-\alpha-\gamma}v < h$. On distingue alors plusieurs cas :

1. $\beta - \alpha - \gamma > 0$: alors $h > 2v$ et v figure dans la tranche en cours donc h n'appartient pas à cette tranche.
2. $\beta - \alpha - \gamma = 0$: alors on a trouvé un élément de la tranche en cours minorant h.
3. $\beta - \alpha - \gamma < 0$: le minorant trouvé pour h est soit non entier soit dans une tranche précédente, on n'a pas obtenu d'information utile.

Dans les bons cas on obtient soit l'information que h est dans une tranche supérieure et donc celle en cours est terminée, soit un minorant probalemement assez proche de h. On peut de même trouver dans les bons cas un majorant de h en considérant le successeur de t, et un autre couple minorant-majorant si h est aussi divisible par 5. Enfin on dispose d'un troisième minorant, le dernier élément inséré dans la tranche en cours avant h ou le début de cette tranche.

Si l'un des majorants est le successeur immédiat de l'un des minorants alors on a trouvé la position d'insertion de h sans avoir fait de comparaison entre entiers de Hamming. Sinon, on peut comparer h au dernier élément de la tranche (2^k) pour savoir si h appartient à cette tranche puis, dans l'affirmative, rechercher la position d'insertion de h par balayage entre l'un des minorants et l'un des majorants.

Expérimentalement cette technique de classement ne donne lieu qu'à environ 400 comparaisons entre entiers de Hamming pour la recherche du millionnième nombre de Hamming, 500 pour la recherche du milliardième, c'est à dire suffisament peu pour que le coût de ces comparaisons soit négligeable dans l'exécution de l'algorithme. On pourrait d'ailleurs le diminuer encore en testant avec les notations précédentes que $\beta \geqslant \alpha$ et en reculant ou avançant de plusieurs positions autour de t pour réaliser cette condition. On tombe alors à 63 comparaisons pour le millionnième nombre de Hamming et à 104 pour le milliardième mais le temps d'exécution n'est pas notablement modifié.

Implémentation (voir le listing page 29) : un entier de Hamming est représente par ses trois exposants et une batterie de pointeurs :

- un pointeur vers l'entier suivant et un vers l'élément précédent pour pouvoir avancer ou reculer à l'intérieur d'une tranche.
- un pointeur vers son tiers, un vers son cinquième, un vers son triple et un vers son quintuple pour accéder rapidement à ces élements.
- un pointeur vers le prochain entier de Hamming impair, utilisé lors de la génération de ceux-cis.

De plus, l'exposant de 2 n'est pas codé directement, on code la différence avec l'exposant du dernier élément de la tranche (2^k) car cette différence est inchangée lorsqu'on multiplie tous les éléments d'une tranche par 2 donc on économise ainsi la mise à jour des exposants, il suffit de conserver dans une variable auxilliaire l'exposant du dernier élément.

exécution :

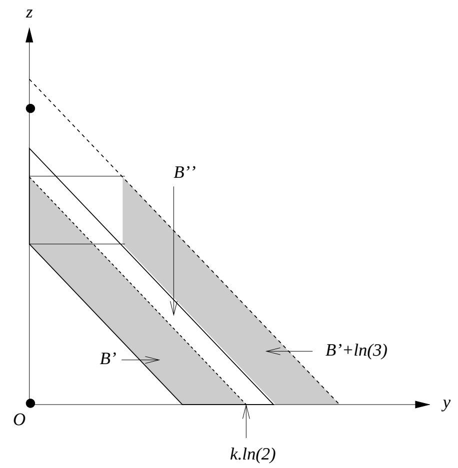
```
bash$ time -f "temps=%es" ./compte-classe 1000000
2^38 . 3^109 * 5^29
temps=0.22s
```

mesures de temps et mémoire :

1	5	10	50	100	500	1000	n (millions)
0.22	0.59	0.97	3.13	5.14	15.88	26.60	temps (s)
1 140	2 068	2 936	7 300	11 028	31 376	49 588	mémoire (Ko)

Complexité : expérimentalement la méthode de Damien tourne en temps $O(n^{2/3})$ ce qui pourrait être justifié en prouvant que l'insertion d'un nouvel entier de Hamming impair se fait en temps borné. Une telle preuve reste à écrire. La complexité spatiale est la taille maximale d'une tranche, soit $O(n^{2/3})$.

3.4 Arbres différentiels



Le principe de translation des tranches de Damien Doligez s'applique aussi en dimension 2 : les entiers de Hamming impairs compris entre 2^{k-1} et 3.2^{k-1} forment une bande dans le plan yOz qui s'obtient à partir de la bande précédente par translation d'une unité parallèlement à l'axe des y et adjonction éventuelle d'une puissance de 5. Cette remarque est à l'origine d'un nouvel algorithme de comptage :

Supposons disposer des entiers de Hamming impairs compris dans l'intervalle $]2^{k-1}, 3.2^{k-1}]$, classés dans une structure de données particulière permettant d'implémenter efficacement les opérations d'insertion, de découpage par rapport à un pivot, de réunion sans recouvrement, de comptage et de translation d'exposants. On note B_k la bande contenant ces entiers. Alors on passe de B_k à B_{k+1} par découpage de B_k en deux parties : $B_k = B'_k \cup B''_k$ avec $B'_k = B_k \cap]2^{k-1}, 2^k]$ et $B''_k = B_k \cap]2^k, 3.2^{k-1}]$, translation de B'_k d'une unité en y ce qui donne formellement $B'_k + \ln 3$, insertion éventuelle d'une puissance de 5 et fusion sans recouvrement avec B''_k. Ceci permet de déterminer de manière incrémentale :

- le cardinal de B'_k ;
- la k-ème tranche de Hamming (tous les entiers de Hamming compris entre 2^{k-1} et 2^k) : $T_k = (T_{k-1} + \ln 2) \cup B'_k$ ainsi que son cardinal ;
- le cardinal $n(2^k)$ du k-ème tétraèdre de Hamming.

Notons que T_k est obtenue comme réunion *avec recouvrement* de translatées des B'_i ce qui permet d'obtenir facilement son cardinal mais plus difficilement son élément de rang $n(2^k) - n$ par ordre décroissant (la méthode de classement de Damien Doligez réalisait à la fois la réunion et le classement de T_k).

Structure de données pour les B_k : on utilise un arbre binaire de recherche ce qui permet, si l'arbre est équilibré, d'effectuer l'insertion le découpage et la réunion sans recouvrement en temps logarithmique (pour la réunion on extrait le plus grand élément de l'arbre de gauche et on le prend comme racine de l'arbre réunion). Le comptage peut être réalisé en temps constant si l'on stocke dans chaque nœud le cardinal du sous-arbre issu de ce nœud et si l'on met à jour ce cardinal lors des opérations d'insertion, découpage et réunion. Enfin, la translation d'exposant peut être réalisée en temps constant si l'étiquette d'un nœud code les exposants du point associé par différence avec les exposants du père. Il suffit alors de translater les exposants de la racine pour translater tout l'arbre, là aussi la gestion de ce codage différentiel est intégrable dans les opérations d'insertion, découpage et réunion. Si les arbres obtenus sont équilibrés, toutes les opérations requises peuvent être effectuées en temps constant ou logarithmique donc le passage d'une tranche de Hamming à la suivante a une complexité en temps $O(\ln n)$ et le calcul complet des tranches successives jusqu'à trouver celle contenant le n-ème nombre de Hamming s'effectue en temps $O(n^{1/3} \ln n)$. L'équilibre des arbres n'est cependant pas garanti, on pourrait maintenir l'équilibre en temps logarithmique par rotations puisqu'on dispose des cardinaux des branches, ceci n'a pas été fait dans le programme listé page 32 par souci de simplification.

Recherche du n-ème nombre de Hamming dans la dernière tranche : cette dernière tranche se présente comme la réunion de $O(n^{1/3})$ bandes triées, on peut obtenir son élément de rang $n(2^k) - n$ par segmentation (on choisit un pivot, on découpe toutes les bandes par rapport à ce pivot en comptant combien d'éléments sont de chaque côté ce qui permet de savoir si le n-ème entier de Hamming se trouve dans une demi-bande gauche, dans une demi-bande droite ou est égal au pivot ; dans les deux premiers cas on itère la recherche sur les demi-bandes correspondantes). Pour déterminer le pivot servant à découper toutes les bandes on commence par isoler une bande de cardinal maximal et on recherche dans cette bande l'élément de rang $\lfloor rc/t \rfloor$ où r est le rang cherché, c le cardinal de la bande et t le cardinal de la tranche. Cette technique s'apparente à la méthode d'interpolation linéaire mais elle n'en a pas les mêmes performances car après quelques découpages les bandes deviennent très petites et le pivot sélectionné peu pertinent. Une méthode plus efficace serait de calculer le maximum et le minimum des éléments restant en lice et de calculer un pivot par interpolation linéaire entre ces extrémums, mais l'ensemble des points de Hamming n'est pas stable par barycentrage et l'utilisation d'un pivot non entier (en calculant le barycentre des logarithmes) complique la validation des comparaisons entre deux nombres de Hamming ou un nombre de Hamming et un pivot ce qui fait que le gain réalisé sur le nombre d'itérations est perdu par des comparaisons plus lentes.

Comparaison entre entiers de Hamming : à la différence de la méthode de comptage et classement simultanés, la méthode décrite ci-dessus donne lieu à un grand nombre de comparaisons lors des découpages donc il importe d'implémenter efficacement ces comparaisons. La comparaison approchée de deux entiers de Hamming par l'intermédiaire de leurs logarithmes est séduisante mais sujette à contestations, ici on l'améliore en effectuant un contrôle de précision lors d'une comparaison et en basculant vers une comparaison exacte en cas de doute. Expérimentalement, pour trouver le 10^p-ème nombre de Hamming il suffit de disposer d'encadrements de $\ln(3)/\ln(2)$ et $\ln(5)/\ln(2)$ à $p+3$ décimales pour ne pas rencontrer de cas douteux.

exécution (les lignes `k=.. t=..` donnent pour chaque étape de segmentation le rang cherché et le nombre d'élements restant en lice) :

```
bash$ time -f "temps=%es" ./difftree 1G
k=1046828 t=1069703
k=1188 t=24063
k=1173 t=24048
k=1091 t=23966
k=1067 t=23942
k=57 t=22932
k=57 t=82
k=57 t=71
k=57 t=60
k=21 t=24
k=5 t=8
k=0 t=3
2^761 . 3^572 . 5^489
temps=3.27s
```

mesures de temps et mémoire :

1	5	10	50	100	500	1000	n (millions)
0.24	0.49	0.65	1.06	1.54	3.01	3.27	temps (s)
940	1 232	1 424	1 944	2 468	4 088	4 848	mémoire (Ko)

Complexité probable : le temps de construction de la tranche de Hamming est $O(n^{1/3}\ln n)$ en supposant que les arbres différentiels restent équilibrés et le temps d'une segmentation est $O(n^{1/3}\ln n)$ également. En supposant que la méthode de segmentation retenue conduit à $O(\ln n)$ itérations, on obtient une complexité temporelle pour le calcul du n-ème nombre de Hamming en $O(n^{1/3}(\ln n)^2)$, l'unité de temps étant une opération entre entiers, mais cette majoration repose sur des hypothèses difficiles à établir. Par ailleurs, si les approximations de $\ln(3)/\ln(2)$ et $\ln(5)/\ln(2)$ à $O(\ln n)$ chiffres sont effectivement suffisantes pour éviter les calculs exacts alors le coût d'une opération entre entiers est $O((\ln n)^2)$ d'où une complexité probable $O(n^{1/3}(\ln n)^4)$. Remarquons que la complexité spatiale est sous les hypothèses précédentes $O(n^{1/3}(\ln n)^3)$ alors qu'on a au cours du calcul engendré une tranche de cardinal $O(n^{2/3})$...

3.5 Méthode volumétrique

Cette méthode est la plus performante de toutes celles présentées, elle fournit le n-ème entier de Hamming en temps $O(n^{1/3}(\ln n)^2)$ et en mémoire logarithmique (l'unité de temps est une opération entre entiers et l'unité mémoire un entier ; si l'on tient compte de ce que les entiers ont une taille machine de l'ordre de $\ln n$ alors la complexité temporelle est $O(n^{1/3}(\ln n)^4)$ et la complexité mémoire $O((\ln n)^2)$). Sur un micro-ordinateur bas de gamme on peut ainsi déterminer le 10^{18}-ème entier de Hamming en moins de deux heures, ce qui est impossible avec les méthodes précédentes pour cause de manque de mémoire.

Le principe est le suivant : pour a, b, c réels strictement positifs et N réel quelconque on note

$$\begin{aligned} S(a,b,N) &= \operatorname{card}\{(x,y)\in\mathbb{N}^2 \ / \ ax+by\leqslant N\} \\ V(a,b,c,N) &= \operatorname{card}\{(x,y,z)\in\mathbb{N}^3 \ / \ ax+by+cz\leqslant N\}. \end{aligned}$$

$S(a,b,N)$ est la «surface discrète» du triangle limité par les deux axes de coordonnées et la droite d'équation $ax+by=N$, $V(a,b,c,N)$ est le «volume discret» du tétraèdre limité par les trois plans de coordonnées et le plan d'équation $ax+by+cz=N$. Ces quantités seront appelées «surface» et «volume» dans la suite. On verra dans la section suivante qu'on peut calculer $S(a,b,N)$ lorsque a et b sont entiers en temps et en mémoire $O(\max(\ln a, \ln b))$ et donc $V(a,b,c,N)$ en temps $O(N\max(\ln a,\ln b)/c)$ d'après la relation :

$$V(a,b,c,N) = \sum_{0\leqslant z\leqslant N/c} S(a,b,N-cz).$$

Dans l'optique de la recherche du n-ème entier de Hamming, a, b, c seront entiers de l'ordre de $10^4 n$ et N entier de l'ordre de $10^4 n^{4/3}$ donc les calculs de surface se font en temps $O(\ln n)$ et les calculs de volume en temps $O(n^{1/3}\ln n)$.

Soient a, b, c entiers strictement positifs tels que

$$\frac{b}{a}\leqslant\frac{\ln 3}{\ln 2}<\frac{b+1}{a}, \qquad \frac{c}{a}\leqslant\frac{\ln 5}{\ln 2}<\frac{c+1}{a}.$$

Donc pour tout K :

$$V\Big(a,b+1,c+1,\frac{Ka}{\ln 2}\Big)\leqslant V(\ln 2,\ln 3,\ln 5,K)\leqslant V\Big(a,b,c,\frac{Ka}{\ln 2}\Big)$$

car les triangles obliques associés aux deux tétraèdres extrêmes «encadrent» le triangle oblique du tétraèdre du milieu. Si ce dernier triangle ne contient aucun point entier, ce qui est le cas entre autres si K n'appartient pas au groupe additif engendré par $\ln 2$, $\ln 3$ et $\ln 5$, alors les trois membres sont égaux pour a suffisamment grand à K fixé donc on peut calculer le volume discret d'un tétraèdre de Hamming donné à condition de disposer d'encadrements suffisamment précis de $\ln 3/\ln 2$ et $\ln 5/\ln 2$. Pour fixer les idées, on choisit K de sorte que $V(\ln 2, \ln 3, \ln 5, K) \approx n$ soit $K \approx \alpha n^{1/3}$ avec $\alpha = (6 \ln 2 \ln 3 \ln 5)^{1/3}$ puis a de sorte que le volume physique compris entre les tétraèdres de paramètres $a, b+1, c+1, Ka/\ln 2$ et $a, b, c, Ka/\ln 2$ soit de l'ordre de 10^{-4} (ceci ne garantit pas le fait que cette région ne contienne pas de points entiers mais le rend plausible) ce qui donne après calculs $a \approx 10^4 \beta K^3$ avec $\beta = (\ln 3 + \ln 5)/(6 \ln^2 3 \ln^2 5)$ donc a, b, c sont de l'ordre de $10^4 n$ et $N = Ka/\ln 2$ de l'ordre de $10^4 n^{4/3}$ comme annoncé.

Avec a, b, c ainsi choisis on détermine par encadrements successifs deux entiers p et q tels que $V(a, b, c, p) \leqslant n < V(a, b, c, q) \leqslant V(a, b, c, p) + k$ où k est une petite constante ($k = 12$ dans l'implémentation décrite ci-après), ceci peut être effectué avec $O(\ln n)$ calculs de volume donc en temps $O(n^{1/3}(\ln n)^2)$, puis on vérifie que $V(a, b+1, c+1, p) = V(a, b, c, p)$ et $V(a, b+1, c+1, q) = V(a, b, c, q)$. Si le test est positif alors on est sûr que le n-ème point de Hamming est compris entre les plans d'équation $ax + by + cz = p$ et $ax + by + cz = q$ et que c'est le point de rang $n - V(a, b, c, p)$ dans la liste ℓ des points à coordonnées entières positives compris entre ces deux plans. On constitue ℓ par balayage en z : pour chaque z entier compris entre 0 et $\lfloor q/c \rfloor$ on compare les surfaces $S(a, b, p - cz)$ et $S(a, b, q - cz)$; la différence entre ces deux nombres est égale au nombre de points de ℓ dont l'exposant de 5 vaut z. Lorsque cette différence est strictement positive on recherche les exposants x et y de ces points par dichotomie, le nombre de points tels que $y > y_0$ est égal à $S(a, b, q - cz - by_0 - b) - S(a, b, p - cz - by_0 - b)$ donc calculable en temps logarithmique et une fois y connu les points de ℓ ayant un exposant de 3 égal à y sont ceux d'abscisse $x_0 = \lfloor (q - cz - by)/a \rfloor$, $x_0 - 1, \ldots$ Enfin, une fois ℓ connue, il ne reste plus qu'à la classer pour l'ordre induit par la forme linéaire $(x, y, z) \mapsto x \ln 2 + y \ln 3 + z \ln 5$ (en effectuant des comparaisons approchées avec contrôle de précision) et à extraire l'élément de rang $n - V(a, b, c, p)$, cet élément est le n-ème entier de Hamming.

La complexité de la phase de constitution de ℓ est $O(n^{1/3} \ln n)$ pour le balayage en z et $O((\ln n)^2)$ pour l'identification de chacun de éléments de ℓ à z déterminé, soit au total $O(n^{1/3} \ln n)$. On a ainsi trouvé le n-ème entier de Hamming en temps $O(n^{1/3}(\ln n)^2)$ (en nombre d'opérations entières) et en mémoire $O(\ln n)$ (en nombre d'entiers).

Exemple d'exécution pour la recherche de l'entier de Hamming de rang 10^{18} (voir le listing page 37) :

```
bash$ ./volume 1E -debug
 0.02  0.02 Newton   2805474567985183128243159433216 -> 1000002623563795213
516.90 516.88 Newton   2805472114538014809295748381758 -> 999999999997705613
1067.01 550.11 Newton   2805472114540160427841063461191 -> 1000000000000000014
1595.97 528.96 Newton   2805472114540160414748825911744 -> 1000000000000000000
```

```
2112.25 516.28 linéaire 280547211454016041661914556l665 -> 1000000000000000003
2628.43 516.18 contrôle 2805472114540160414748825911744 -> 1000000000000000000
3144.42 515.99 contrôle 2805472114540160416619145561665 -> 1000000000000000003
3660.37 515.95 balayage ...
4713.12 1052.75 terminé

x=625963 y=1360652 z=9874
```

Les deux premières colones indiquent en secondes le temps total écoulé depuis le début de l'exécution et la différence avec le temps précédent c'est à dire le temps de la phase précédente. La troisième colone indique quelle est la phase en cours et les colones suivantes donnent des détails sur cette phase.

- Newton $N \to V(a,b,c,N)$: raffinement de l'encadrement (p,q) par la méthode de Newton en considérant que $V(a,b,c,N) \approx N^3/6abc$ pour le calcul de la dérivée. L'encadrement initial est fixé à $p = 0, q = \sqrt[3]{6abcn}$.
- linéaire $N \to V(a,b,c,N)$: on termine la phase de raffinement par interpolation linéaire lorsque l'un des volumes calculés est suffisament proche de n ou lorsque la méthode de Newton ne resserre plus l'encadrement précédent. Dans l'exemple $n = 10^{18}$ il a fallu quatre itérations de Newton, soit quatre calculs de volume pour trouver un minorant p raisonable, et une itération par interpolation linéaire, donc un cinquième calcul de volume, pour trouver q tel que $V(a,b,c,q) - V(a,b,c,p) \leqslant 12$. On aurait économisé un calcul de volume, soit environ 9 minutes si l'on avait fixé la taille maximale de ℓ un tantinet plus haut...
- contrôle $N \to V(a,b+1,c+1,N)$: on vérifie l'encadrement trouvé.
- balayage : phase de recherche des trois éléments de ℓ par balayage en z. Expérimentalement cette phase a un coût en temps de l'ordre de deux calculs de volume ce qui s'explique par la méthode utilisée, on calcule pour chaque z la différence entre deux surfaces.
- terminé : au bout de 4713.12 secondes exactement. Le 10^{18}-ème entier de Hamming a pour exposants 625 963, 1 360 652 et 9 874 donc s'écrit en base 10 avec 844 532 chiffres dont 9 874 zéros en queue...

mesures de temps et mémoire :

10^6	10^9	10^{12}	10^{15}	10^{18}	n
0.34	3.17	24.62	339.49	4 713.12	temps (s)
808	836	1 044	1 044	1 044	mémoire (Ko)

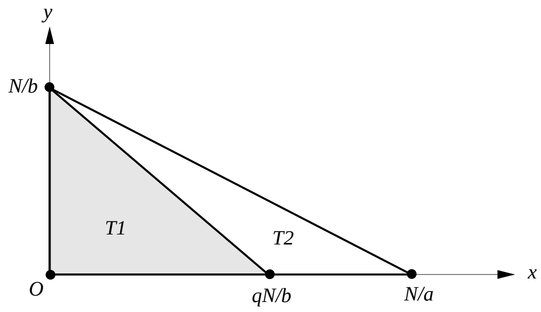
3.5.1 Calcul de surface

Soient $a, b \in \mathbb{N}$ avec $0 < a < b$ et N entier ; on veut calculer

$$S(a,b,N) = \text{card}\,\{(x,y) \in \mathbb{N}^2 \ / \ ax + by \leqslant N\}.$$

Déjà on peut remarquer que $S(a,b,N) = 0$ si $N < 0$ et $S(a,b,N) = 1 + \lfloor N/a \rfloor$ si $N < b$ (triangle de hauteur strictement inférieure à 1). Dans le cas général, on effectue la division euclidienne de b par a, soit $b = qa + r$, et on divise le triangle à mesurer en deux parties :

$$T_1 : x \geqslant 0, \quad y \geqslant 0, \quad \frac{b}{q}x + by \leqslant N$$
$$T_2 : y \geqslant 0, \quad \frac{b}{q}x + by > N, \quad ax + by \leqslant N.$$



La surface discrète de T_1 est calculable directement :

$$S(T_1) = \sum_{y=0}^{\lfloor N/b \rfloor} 1 + \Big\lfloor \frac{qN}{b} \Big\rfloor - qy = \Big(1 + \Big\lfloor \frac{N}{b} \Big\rfloor\Big)\Big(1 + \Big\lfloor \frac{qN}{b} \Big\rfloor - \frac{q}{2}\Big\lfloor \frac{N}{b} \Big\rfloor\Big).$$

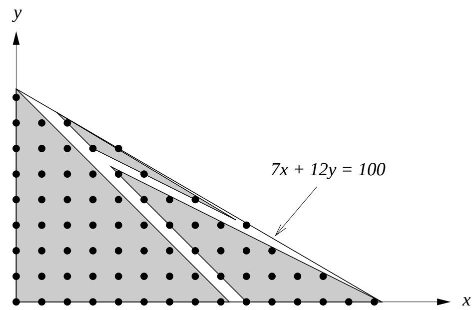
Pour calculer la surface de T_2 on effectue le changement de variable $t = x + qy$ soit

$$T_2 : y \geqslant 0, \quad t > \frac{qN}{b}, \quad at + ry \leqslant N$$

et donc

$$S(T_2) = S\Big(r, a, N - \Big\lfloor \frac{qN}{b} \Big\rfloor - a\Big).$$

Ceci donne donc une relation de récurrence sur S où les paramètres a, b, N sont transformés en r, a et $N' < N$. Le nombre d'étapes avant d'arriver à un cas de base (hauteur < 1) est au plus égal au nombre d'étapes nécessaires au calcul du pgcd de a et b (sinon on aboutit au calcul de $S(0, d, N'')$ avec $N'' > 0$ ce qui donnerait une surface infinie) donc est $O(\max(\ln a, \ln b))$. La figure ci-dessous illustre les divisions effectuées dans le cas $a = 7, b = 12, N = 100$:



Complexité mémoire : le calcul récursif de $S(a, b, N)$ peut être conduit en mémoire constante puisqu'il se réduit à une accumulation. Toutefois, comme on sera amené à calculer de nombreuses surfaces ayant les mêmes paramètres a et b, il est avantageux en termes de temps de calcul de mémoriser la suite des paramètres des formules closes ce qui conduit à un encombrement mémoire en $O(\max(\ln a, \ln b))$.

3.5.2 Calcul de volume

La technique de découpage des triangles vue à la section précédente se transpose mal en dimension 3. On peut découper un tétraèdre en deux parties dont le volume de l'une est calculable par une formule close mais la partie restante n'est pas un tétraèdre, elle peut être décomposée en différence de deux tétraèdres ce qui donne une relation de récurrence compliquée pour laquelle la convergence vers un cas de base n'est pas triviale et en tous cas pas rapide. Expérimentalement, la méthode de découpage décrite ci-dessous donne un temps de calcul de volume en $O(n^{1/2})$ donc est moins performante que le simple découpage d'un tétraèdre en $O(n^{1/3})$ tranches dont on calcule les surfaces en temps $O(\ln n)$. Toutefois, si on arrête la récursion à une petite profondeur (5 semble optimal) et si on calcule les volumes des tétraèdres obtenus par addition de surfaces on obtient une méthode de calcul de volume environ deux fois plus rapide que l'addition des surfaces de toutes les tranches du tétraèdre original.

Soient a, b, c, N entiers avec $0 < a \leqslant b \leqslant c$ et $T = OABC$ le tétraèdre limité par les plans de coordonnées et le plan d'équation $ax + by + cz = N$.

Formule close

Comme en dimension 2, le volume de T est donné par une formule close lorsque les coefficients a, b, c ont des rapports entiers l'un à l'autre. Plus précisément, pour q, q' et N entiers :

$$\begin{aligned}
V(1, q', qq', N) &= \sum_{z=0}^{\lfloor N/qq' \rfloor} S(1, q', N - qq'z) \\
&= \sum_{z=0}^{\lfloor N/qq' \rfloor} (\lfloor N/q' \rfloor - qz + 1)(N - qq'z - \frac{q'}{2}(\lfloor N/q' \rfloor - qz) + 1) \\
&= \frac{1}{12}(\gamma + 1)(\gamma(2\gamma + 1)q^2 q' - 3\gamma q(2N + q' + 2) + 6\beta(2(N + 1) - (1 + \beta)q') + 12(N + 1))
\end{aligned}$$

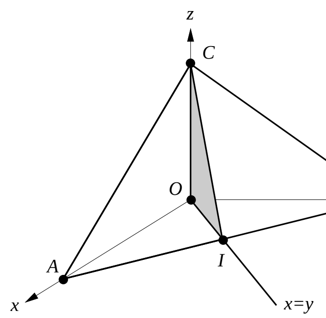
avec $\beta = \lfloor N/q' \rfloor$ et $\gamma = \lfloor N/qq' \rfloor$. Lorsque $q' = 1$, cette formule se simplifie en :

$$V(1, 1, q, N) = \frac{1}{12}(\gamma + 1)(\gamma(2\gamma + 1)q^2 - 3\gamma q(2N + 3) + 6(N + 1)(N + 2)).$$

Cas symétrique

Si $a = b$ alors T est symétrique par rapport au plan d'équation $x = y$ et peut donc être divisé en deux tétraèdres de même volume en comptant à part la tranche OCI située dans le plan bissecteur ce qui donne la relation :

$$V(a, a, c, N) = 2V(a, 2a, c, N - 2a) + S(2a, c, N).$$



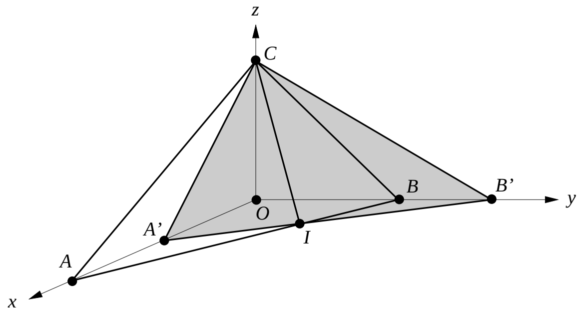
Cas asymétrique croisé

Si $a < b$ et s'il existe un sous-multiple de c compris entre a et b : $a < c/q < b$ alors on considère le tétraèdre $T' = OA'B'C'$ limité par le plan d'équation

$$\frac{c}{q}x + \frac{c}{q}y + cz = N$$

dont le volume est $V(T) = V(1, 1, q, \lfloor qN/c \rfloor)$ connu. Les faces obliques de T et T' se croisent le long du segment IC et :

$$V(T) = V(T') + V(A'AIC) - V(B'BIC).$$



$A'AIC$ est défini par les inéquations

$$y \geqslant 0, \qquad z \geqslant 0, \qquad x + y + qz > \frac{qN}{c}, \qquad ax + by + cz \leqslant N$$

soit en posant $t = x + y + qz - \left\lfloor \frac{qN}{c} \right\rfloor - 1$

$$y \geqslant 0, \qquad z \geqslant 0, \qquad t \geqslant 0, \qquad at + (b - a)y + (c - qa)z \leqslant N - a\left\lfloor \frac{qN}{c} \right\rfloor - a$$

et donc

$$V(A'AIC) = V(a, b - a, c, N - a\lfloor qN/c \rfloor - a).$$

Des calculs similaires donnent

$$V(B'BIC) = V(b-a, b, qb-c, b\lfloor qN/c \rfloor - N - 1)$$

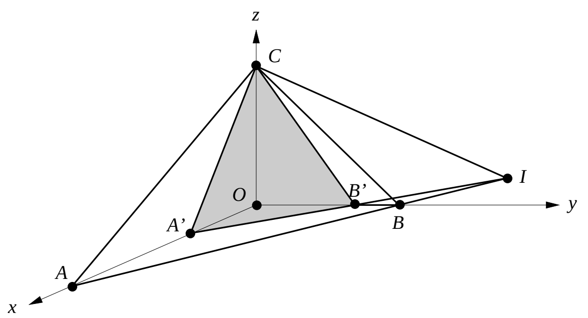
d'où finalement

$$\begin{aligned} V(a,b,c,N) &= V(1,1,q,\lfloor qN/c \rfloor) \\ &+ V(a, b-a, c-qa, N - a\lfloor qN/c \rfloor - a) \\ &- V(b-a, b, qb-c, b\lfloor qN/c \rfloor - N). \end{aligned}$$

Cas asymétrique emboîté

Si $a < b$ et s'il n'existe pas de sous-multiple de c séparant a et b alors on considère un sous-multiple $c/q < a$ et le tétraèdre T' défini comme dans le cas précédent, mais ici T' est inclus dans T. Les prolongements des faces obliques de T et T' se coupent le long d'un segment CI et l'on a

$$\begin{aligned} V(a,b,c,N) &= V(1,1,q,\lfloor qN/c \rfloor) + V(A'AIC) - V(B'BIC) \\ V(A'AIC) &= V(a, b-a, c-qa, N - a\lfloor qN/c \rfloor - a) \\ V(B'BIC) &= V(b-a, b, c-qb, N + a - b\lfloor qN/c \rfloor - 2b). \end{aligned}$$



Convergence

La quantité $\max(a,b,c)$ décroît strictement dans le cas $a < b$ et décroît strictement en deux étapes dans le cas $a = b$ donc la terminaison du calcul est assurée et la profondeur de récursion est majorée par $2c$, ce qui donne un temps de calcul en $O(2^{2c})$ avec c de l'ordre de $100n$... En pratique le temps de calcul n'est pas aussi catastrophique et le fait de borner la profondeur de récursion puis d'additionner les surfaces des tranches des tétraèdres obtenus permet de maintenir une complexité temporelle $O(n^{1/3} \ln n)$.

Chapitre 4

Conclusion

Le meilleur algorithme pour calculer tous les entiers de Hamming jusqu'au n-ème est donc celui de Damien Doligez et il est probablement imbattable : aucun nombre en trop n'est calculé, accélérer les quelques comparaisons qui sont effectuées en utilisant les logarithmes aura un effet négligeable sur le temps global de calcul. Pour rendre justice à Denis Cazor, je signale que son programme initial était rédigé en Pascal, ce qui, après traduction en C donne un temps de calcul de 2.47 secondes sur la machine de test pour le millionième nombre de Hamming. Il bat donc celui de Damien mais ceci uniquement grâce aux meilleures performances du compilateur C (`gcc`) par rapport au compilateur Ocaml (`ocamlopt`).

En ce qui concerne le calcul direct du n-ème entier de Hamming la méthode volumétrique est la plus efficace connue à ce jour tant en termes de temps de calcul que de mémoire requise, et permet le calcul du n-ème entier de Hamming en temps raisonnable pour des valeurs de n extrêmement élevées. Cependant la situation n'est pas intellectuellement satisfaisante, on attend logiquement un nouvel algorithme fournissant le n-ème entier de Hamming en temps polylogarithmique. Remarquons qu'il existe un tel algorithme pour le problème de Hamming restreint à deux facteurs premiers puisqu'on peut calculer des surfaces de triangles en temps logarithmique. La technique de division des tétraèdres donne des résultats plutôt décevants en comparaison de ceux obtenus par division de triangles et le calcul du volume d'un tétraèdre semble être un problème nettement plus complexe que son équivalent bidimensionnel. Signalons que ce problème admet une solution logarithmique lorsque les sommets du tétraèdre sont à coordonnées entières (voir L.J. Mordell, *Lattice points in a tetrahedron and generalized Dedekind sums,* Journal Indian Mathem. Soc. 15 (1951), pp 41-46), mais les tétraèdres de Hamming ou leurs approximations rationnelles ne sont malheureusement pas à sommets entiers.

Chapitre 5

Code des programmes

5.1 Listes paresseuses

```
(* fichier listp.ml : calcul du n-ème entier de Hamming par liste paresseuse *)

open Lazy
open Num

(* liste paresseuse d'entiers de Hamming *)
type listp = {hd:num; tl:listp Lazy.t}

(* n-ème élément d'une liste paresseuse *)
let rec nème n liste = if n = 0 then liste.hd else nème (n-1) (force liste.tl)

(* multiplie une liste par p *)
let rec mult p liste =
  {hd = liste.hd */ p;
   tl = lazy(mult p (force liste.tl))}

(* fusion sans répétition de listes paresseuses *)
let rec (@) a b = let c = compare_num a.hd b.hd in
       if c < 0 then {hd=a.hd; tl=lazy((force a.tl) @     b       )}
  else if c > 0 then {hd=b.hd; tl=lazy(    a        @ (force b.tl))}
  else               {hd=a.hd; tl=lazy((force a.tl) @ (force b.tl))}

(* calcul *)
let _ =
  let rec h =
    {hd = (Int 1);
     tl = lazy((mult (Int 2) h) @ (mult (Int 3) h) @ (mult (Int 5) h))} in
  let n = int_of_string(Sys.argv.(1)) in
  print_string(string_of_num(nème n h));
  print_newline()
```

5.2 Files de priorité

```
(* fichier prifile.ml : insertion dans une file de priorité *)
```

```
(* représentation de 2^a*3^b*5^c, l=logarithme *)
type entier = {a:int; b:int; c:int; l:float}
let ln2 = log(2.0)
and ln3 = log(3.0)
and ln5 = log(5.0)

(* file de priorité = ensemble ordonné *)
module Hset = Set.Make(
  struct
    type t = entier
    let compare x y = Pervasives.compare x.l y.l
  end)
open Hset

(* avance de n éléments *)
let rec avance n h =
  let x = min_elt h in
  if n=0 then x
  else
    let h = remove x h in
    let h = add {x with c=x.c+1; l=x.l+.ln5} h in
    let h = if x.c=0         then add {x with b=x.b+1; l=x.l+.ln3} h else h in
    let h = if x.c=0 & x.b=0 then add {x with a=x.a+1; l=x.l+.ln2} h else h in
    avance (n-1) h

(* lance les calculs *)
let _ =
  let h = singleton {a=0; b=0; c=0; l=0.} in
  let n = int_of_string(Sys.argv.(1)) in
  let t = avance n h in
  Printf.printf "2^%d . 3^%d . 5^%d" t.a t.b t.c;
  print_newline()
```

5.3 Listes mutables

```
(* fichier cazor.ml : insertion en milieu de liste *)

(* a,b,c = exposants, l=logarithme *)
type listm = {a:int; b:int; c:int; l:float; mutable tl:listm}
let ln2 = log(2.)
and ln3 = log(3.)
and ln5 = log(5.)
let rec nil={a=0; b=0; c=0; l=0.; tl=nil} (* fin de liste *)

(* insère la cellule u à la bonne place celle après pointée par x *)
(* et retourne le nouveau pointeur d'insertion *)
let rec insère u x =
  if u.l > x.tl.l then insère u x.tl
                  else begin
                    x.tl <- {u with tl=x.tl};
                    x.tl
```

```
                 end

(* avance de n éléments dans la liste de Hamming *)
(* x = dernier double inséré, y = dernier triple, z = dernier quintuple *)
let rec avance n ({a=a; b=b; c=c; l=l} as h) x y z =
  if n=0 then h
  else begin
    let z' = {a=a; b=b; c=c+1; l=l+.ln5; tl=nil} in z.tl <- z';
    let y' = if h.c=0        then insère {h with b=b+1; l=l+.ln3} y else y in
    let x' = if h.c=0 & h.b=0 then insère {h with a=a+1; l=l+.ln2} x else x in
    avance (n-1) h.tl x' y' z'
  end

(* lance les calculs *)
let _ =
  let h = {a=0; b=0; c=0; l=0.; tl=nil} in
  let n = int_of_string(Sys.argv.(1))   in
  let t = avance n h h h h              in
  Printf.printf "2^%d . 3^%d . 5^%d" t.a t.b t.c;
  print_newline()
```

5.4 Classement sans comparaison

```
(* fichier nocomp.ml : classement sans comparaison *)

(* Suite de Hamming : a,b,c sont les exposants de 2,3,5 *)
type listm = {a:int; b:int; c:int; mutable tl:listm}
let (=.) x y = x.a = y.a & x.b = y.b & x.c = y.c

(* multiplication et concaténation en tête *)
let double x    = {a=x.a+1; b=x.b;   c=x.c;   tl=x}
and triple x    = {a=x.a;   b=x.b+1; c=x.c;   tl=x}
and quintuple x = {a=x.a;   b=x.b;   c=x.c+1; tl=x}

(* calcul de minimum (méthode de D. Doligez *)
open Num

let valeur(x) = (Int 2)**/(Int x.a) */ (Int 3)**/(Int x.b) */ (Int 5)**/(Int x.c)
let min x y = if valeur(x) <=/ valeur(y) then x else y

let dmin x y z =
  match (y.a>0,z.b>0,x.c>0,y.c>0,x.b>0,z.a>0    ) with
      (* x<=y  y<=z  z<=x  z<=y  y<=x  x<=z  *)
        |true, _,    _,    _,    _,    true  -> x
        |true, _,    true, _,    _,    _     -> z
        |true, true, _,    _,    _,    _     -> x
        |_,    true, _,    _,    true, _     -> y
        |_,    true, true, _,    _,    _     -> y
        |_,    _,    true, true, _,    _     -> z
        |_,    _,    _,    true, true, _     -> z
        |_,    _,    _,    true, _,    true  -> x
        |_,    _,    _,    _,    true, true  -> y
```

```
        |true, _,    _,    _,    _,    _     -> min x z
        |_,    true, _,    _,    _,    _     -> min x y
        |_,    _,    true, _,    _,    _     -> min y z
        |_,    _,    _,    true, _,    _     -> min x z
        |_,    _,    _,    _,    true, _     -> min y z
        |_,    _,    _,    _,    _,    true  -> min x y

        |_,    _,    _,    _,    _,    _     -> min x (min y z)

(* allonge la liste de Hamming de n éléments *)
let rec allonge n h x y z =
  if n = 0 then h
  else begin
    let t = dmin x y z in
    h.tl <- t;
    allonge (n-1) t (if t =. x then double    x.tl.tl else x)
                    (if t =. y then triple    y.tl.tl else y)
                    (if t =. z then quintuple z.tl.tl else z)
  end

(* lance les calculs *)
let _ =
  let rec h = {a=0; b=0; c=0; tl=h} in
  let n = int_of_string(Sys.argv.(1)) in
  let t = allonge n h (double h) (triple h) (quintuple h) in
  Printf.printf "2^%d . 3^%d . 5^%d" t.a t.b t.c;
  print_newline()
```

5.5 Recherche par comptage

```
(* fichier comptage.ml: recherche par comptage *)

(* exposants de 2,3,5 et logarithme *)
type entier = {a:int; b:int; c:int; l:float};;
let lg3 = log(3.0)/.log(2.0)
and lg5 = log(5.0)/.log(2.0)

(* cherche le plus petit k tel que n(2^k) >= n *)
(* retourne n(2^k)-n et k                      *)
let rec cherche n k =
  if n <= 0 then (-n,k)
  else begin
    let w = ref(float(k+1))
    and c = ref(n) in
    while !w >= 0.0 do c := !c - truncate(!w/.lg3) - 1; w := !w -. lg5 done;
    cherche !c (k+1)
  end

(* liste triée des points dans la tranche ]2^(k-1),2^k] *)
let liste(k) =
  let res = ref [] in
```

```
  for z=0 to truncate(float(k)/.lg5) do
    for y=0 to truncate((float(k)-.float(z)*.lg5)/.lg3) do
      let l = float(y)*.lg3 +. float(z)*.lg5 in
      let x = truncate(float(k) -. l)        in
      res := {a=x; b=y; c=z; l = float(x) +. l} :: !res
    done
  done;
  Sort.list (fun u v -> u.l > v.l) !res


(* lance les calculs *)
let _ =
  let n = int_of_string(Sys.argv.(1)) in
  let (p,k) = cherche n 0      in
  let t = List.nth (liste k) p in
  Printf.printf "2^%d . 3^%d . 5^%d" t.a t.b t.c;
  print_newline()
```

5.6 Comptage et classement simultanés

```
(* fichier compte-classe.ml : comptage et classement simultanés *)
open Num

type cellule = {
  mutable a:int; b:int; c:int;   (* exposants de 2,3,5 *)
  mutable tiers     : cellule;   (* lien vers u/3 *)
  mutable triple    : cellule;   (* lien vers 3u  *)
  mutable cinquième : cellule;   (* lien vers u/5 *)
  mutable quintuple : cellule;   (* lien vers 5u  *)
  mutable précédent : cellule;   (* lien arrière dans la suite 2-3-5 *)
  mutable suivant   : cellule;   (* lien avant dans la suite 2-3-5   *)
  mutable suiv35    : cellule    (* lien avant dans la suite 3-5     *)
}
let rec nil =
  {a=0;b=0;c=0;
   tiers     = nil; triple    = nil;
   cinquième = nil; quintuple = nil;
   précédent = nil; suivant   = nil;
   suiv35    = nil}

              (* +--------------------------------------------+
                 |  Génération des entiers de Hamming impairs  |
                 +--------------------------------------------+ *)

let troisfois  x = {x with b=x.b+1; suiv35=x}
and cinqfois   x = {x with c=x.c+1; suiv35=x}

let rec début =
  {a=0;b=0;c=0;
   tiers     = nil;   triple    = nil;
   cinquième = nil;   quintuple = nil;
   précédent = début; suivant   = début;
```

```
  suiv35   = nil}


let h = ref(début)        (* dernier élément engendré *)
and y = ref(troisfois début)    (* prochain triple    *)
and z = ref(cinqfois  début)    (* prochain quintuple *)

(* calcul de minimum *)
let dmin y z = match (z.b>0, y.c>0) with
  |true,_    -> y
  |_,   true -> z
  |_,   _    -> if Int(3)**/Int(y.b) */ Int(5)**/Int(y.c) >/
                   Int(3)**/Int(z.b) */ Int(5)**/Int(z.c)
                then z else y

(* prolonge la liste de Hamming 3-5 d'un élément *)
let avance35() =
  let t = dmin !y !z in
  !h.suiv35 <- t;
  h := t;
  if t.b = !y.b & t.c = !y.c then begin
    let u = !y.suiv35 in
    t.tiers  <- u;
    u.triple <- t;
    y := troisfois u.suiv35
  end
  else t.tiers <- nil;
  if t.b = !z.b & t.c = !z.c then begin
    let u = !z.suiv35 in
    t.cinquième <- u;
    u.quintuple <- t;
    z := cinqfois u.suiv35
  end
  else t.cinquième <- nil

                  (* +--------------------------+
                     |  Classement et comptage  |
                     +--------------------------+ *)

let p  = ref(début) (* pointeur d'insertion dans la tranche *)
and a  = ref(-1)    (* décalage exposant de 2 *)
and c1 = ref(0)     (* cardinal des tranches précédentes *)
and c2 = ref(1)     (* cardinal de la tranche courante   *)

(* comparaison exacte *)
let diff x y = if x>y then (x-y,0) else (0,y-x)
let (<<) u v =
  let (a,a') = diff u.a v.a
  and (b,b') = diff u.b v.b
  and (c,c') = diff u.c v.c in
  Int(2)**/Int(a)  */ Int(3)**/Int(b)  */ Int(5)**/Int(c) </
  Int(2)**/Int(a') */ Int(3)**/Int(b') */ Int(5)**/Int(c')

(* insère un élément après la cellule pointée par q *)
```

```
let rec place q =
  !h.suivant <- q.suivant;
  q.suivant.précédent <- !h;
  q.suivant <- !h;
  !h.précédent <- q;
  p := !h;
  incr c2;
  avance35();
  avance235()

(* augmente la tranche courante d'un élément tant que possible *)
(* lève l'exception Exit quand la tranche est pleine           *)
and avance235() =

  !h.a <- - !a; (* mise à niveau exposant de 2 *)

  (* cherche un minorant à partir du tiers *)
  let min3 =
    let t = !h.tiers    in
    let u = t.précédent in
    let v = u.triple    in
    if t == nil or u == début then nil
    else if u.a = t.a + v.a + !a then v
    else if u.a > t.a + v.a + !a then raise Exit
    else nil in

  (* cherche un minorant à partir du cinquième *)
  let min5 =
    let t = !h.cinquième in
    let u = t.précédent  in
    let v = u.quintuple  in
    if t == nil or u == début then nil
    else if u.a = t.a + v.a + !a then v
    else if u.a > t.a + v.a + !a then raise Exit
    else nil in

  (* cherche un majorant à partir du tiers *)
  let max3 =
    let t = !h.tiers  in
    let u = t.suivant in
    let v = u.triple  in
    if t == nil or t == début then nil
    else if u.a = t.a + v.a + !a then v
    else if u.a > t.a + v.a + !a
    then if début << !h then raise Exit else début
    else nil in

  (* cherche un majorant à partir du cinquième *)
  let max5 =
    let t = !h.cinquième in
    let u = t.suivant    in
    let v = u.quintuple  in
    if t == nil or t == début then nil
    else if u.a = t.a + v.a + !a then v
```

```
    else if max3 == nil & u.a > t.a + v.a + !a
    then if début << !h then raise Exit else début
    else nil in

  (* si un majorant suit un minorant c'est bon *)
       if min3 != nil & (min3 == max3.précédent or min3 == max5.précédent) then place min3
  else if min5 != nil & (min5 == max3.précédent or min5 == max5.précédent) then place min5

  (* sinon recule à partir d'un majorant s'il y en a un *)
  else let q = ref(if max3 != nil then max3 else max5) in
    if !q != nil then begin
      while !q.précédent != !p & !q.précédent != min3 & !q.précédent != min5 & !h << !q.précédent
      do q := !q.précédent done;
      place !q.précédent
    end

    (* dernier recours: avance à partir d'un minorant ou de p *)
    else begin
      if min5 != nil then p := min5;
      if min3 != nil then p := min3;
      while !p.suivant != max3 & !p.suivant != max5 & !p.suivant << !h do
        p := !p.suivant;
        if !p == début then raise Exit
      done;
      place !p
    end
(* fin de avance235 (si, si) *)

(* lance les calculs *)
let _ =
  let n = int_of_string(Sys.argv.(1)) in
  avance35();
  while !c1 <= n do
    incr a;
    p := début;
    (try avance235() with Exit -> ());
    c1 := !c1 + !c2;
  done;
  let x = ref(début) in
  for i = !c1-1 downto n+1 do x := !x.précédent done;
  Printf.printf "2^%d . 3^%d * 5^%d" (!x.a + !a) !x.b !x.c;
  print_newline()
```

5.7 Arbres différentiels

```
(* fichier difftree.ml: n-ème nombre de Hamming par arbres différentiels *)

open Num

                 (*+-----------------------------------------+
                   |  Structure abstraite de groupe ordonné  |
                   +-----------------------------------------+*)
```

```
module type Groupe = sig
  type t                     (* type de base   *)
  val (++) : t -> t -> t     (* addition       *)
  val (--) : t -> t -> t     (* soustraction   *)
  val (<<) : t -> t -> bool  (* comparaison    *)
  val zéro : t               (* constantes     *)
  val ln2  : t
  val ln3  : t
  val ln5  : t
  val string : t -> string   (* affichage *)
end

                        (*+------------------------+
                          |  Arbres différentiels  |
                          +------------------------+*)

module Difftree(G:Groupe) : sig

  type arbre                                       (* type abstrait           *)
  val  singleton  : G.t -> arbre                   (* arbre à un élément      *)
  val  taille     : arbre -> num                   (* taille d'un arbre       *)
  val  inc        : G.t -> arbre -> arbre          (* translation             *)
  val  dec        : G.t -> arbre -> arbre          (* translation réciproque  *)
  val  insère     : G.t -> arbre -> arbre          (* ajoute un élément       *)
  val  découpe    : G.t -> arbre -> arbre*arbre    (* découpe pr. à un pivot  *)
  val  recolle    : arbre -> arbre -> arbre        (* fusion sans recouvrement *)
  val  kème_forêt : num -> num -> arbre list -> G.t (* recherche par rang      *)

end = struct
  open G

  (*               cardinal, étiquette, gauche, droite *)
  type arbre = Vide | Noeud of   num * t * arbre * arbre

  (* utilitaires *)
  let singleton a = Noeud(Int 1, a, Vide, Vide)
  and taille = function Vide -> Int 0 | Noeud(t,_,_,_) -> t
  and inc x  = function Vide -> Vide  | Noeud(t,a,g,d) -> Noeud(t,a++x,g,d)
  and dec x  = function Vide -> Vide  | Noeud(t,a,g,d) -> Noeud(t,a--x,g,d)

  (* insère un élément *)
  let rec insère x = function
    | Vide           -> singleton(x)
    | Noeud(t,a,g,d) -> if a << x then Noeud(t+/Int(1), a, g, insère (x--a) d)
                                  else Noeud(t+/Int(1), a, insère (x--a) g, d)

  (* découpe l'arbre en deux parties séparés par x *)
  let rec découpe x = function
  | Vide           -> (Vide,Vide)
  | Noeud(t,a,g,d) -> if a << x
      then let (d1,d2) = découpe (x--a) d in Noeud(t-/taille(d2), a, g, d1), (inc a d2)
      else let (g1,g2) = découpe (x--a) g in (inc a g1), Noeud(t-/taille(g1), a, g2, d)
```

```
(* retire le plus grand élément d'un arbre *)
let rec maximum = function
| Vide -> failwith "max(vide)"
| Noeud(_,a,g,Vide) -> (a,inc a g)
| Noeud(t,a,g,d)    -> let (x,d') = maximum(d) in (x++a),Noeud(t-/Int(1), a, g, d')

(* recolle deux arbres sans recouvrement *)
let recolle g d =
  if g = Vide then d
  else let (a,g') = maximum(g) in Noeud(taille(g)+/taille(d), a, dec a g', dec a d)

(* découpe une forêt en deux forêts séparées par x    *)
(* et retourne les deux forêts et leurs tailles.      *)
(* (t1,t2,f1,f2) = résultat partiel pour accumulation *)
let rec découpe_forêt x ((t1,t2,f1,f2) as accu) = function
  | []          -> accu
  | tête::queue -> let (g,d) = découpe x tête in
                   découpe_forêt x (t1+/taille(g), t2+/taille(d),
                                    (if g = Vide then f1 else g::f1),
                                    (if d = Vide then f2 else d::f2)) queue

(* extrait un arbre de taille maximale *)
let rec max_arbre a f = function
| []       -> (a,f)
| b::suite -> if taille(b) >/ taille(a) then max_arbre b (a::f) suite
                                        else max_arbre a (b::f) suite

(* extrait la k-ème étiquette d'un arbre en partant de la fin *)
let rec kème_arbre k = function
  | Vide -> failwith "arbre vide"
  | Noeud(t,a,g,d) ->
      if taille(d) =/ k then (a, inc a g, inc a d)
      else if taille(d) >/ k
      then let (x,d1,d2) = kème_arbre k d
           in (a++x, Noeud(t-/taille(d2)-/Int(1),a,g,d1), inc a d2)
      else let (x,g1,g2) = kème_arbre (k-/taille(d)-/Int(1)) g
           in (a++x, inc a g1, Noeud(t-/taille(g1)-/Int(1),a,g2,d))

(* extrait la k-ème étiquette de la forêt f en partant de la fin *)
(* t = cardinal de la forêt                                      *)
let rec kème_forêt k t f = match f with
    | [] -> failwith "forêt vide"
    | arbre::suite ->
        Printf.printf "k=%s t=%s\n" (string_of_num k) (string_of_num t);
        let (arbre,suite) = max_arbre arbre [] suite in
        let k' = quo_num (k*/taille(arbre)) t in
        let (a,g,d) = kème_arbre k' arbre in
        let (t1,t2,f1,f2) = découpe_forêt a
                              (taille(g), taille(d),
                               (if g = Vide then [] else [g]),
                               (if d = Vide then [] else [d])) suite in
        if      t2 >/ k then kème_forêt k t2 f2
        else if t2 =/ k then a
        else                 kème_forêt (k-/t2-/Int(1)) t1 f1
```

```
end (* Difftree *)


                  (* +---------------------------+
                     |  n-ème nombre de Hamming  |
                     +---------------------------+ *)

module Hamming(G:Groupe) : sig

  val hamming : num -> G.t

end = struct

  module D = Difftree(G)
  open G
  open D

  type tétraèdre = {
    mutable m2: t;         (* multiple de ln2         *)
    mutable m5: t;         (* multiple suivant de ln5 *)
    mutable n : num;       (* taille tétraèdre        *)
    mutable p : num;       (* taille triangle         *)
    mutable l : arbre;     (* bande d'épaisseur ln3   *)
    mutable t : arbre list (* tranche d'épaisseur ln2 *)
  }

  (* retourne le kème nombre de Hamming *)
  let hamming k =

  let t = {m2=zéro; m5=ln5; n=Int(0); p=Int(0); l=singleton(zéro); t=[]} in
  while t.n </ k do

    let m2 = t.m2 in
    t.m2 <- t.m2 ++ ln2;
    let (g,d) = découpe t.m2 t.l in
    t.t <- dec m2 g :: t.t;
    t.l <- recolle d (inc ln3 g);
    t.p <- t.p +/ taille(g);
    t.n <- t.n +/ t.p;
    if t.m5 << t.m2++ln3 then (t.l <- insère t.m5 t.l; t.m5 <- t.m5 ++ ln5)

  done;

  if t.n =/ k then t.m2
  else let a = kème_forêt (t.n-/k-/Int(1)) t.p t.t in a ++ t.m2 -- ln2

end (* Hamming *)

                    (* +----------------------+
                       |  entiers de Hamming  |
                       +----------------------+ *)

module Hnum : Groupe = struct
```

```
  type t = int * int * int (* exposants de 2 3 5 *)
  let zéro = (0, 0, 0)
  and ln2  = (1, 0, 0)
  and ln3  = (0, 1, 0)
  and ln5  = (0, 0, 1)
  and (++) (a,b,c) (x,y,z) = (a+x,b+y,c+z)
  and (--) (a,b,c) (x,y,z) = (a-x,b-y,c-z)
  and string(a,b,c) = Printf.sprintf "2^%d . 3^%d . 5^%d" a b c

  (* approximations par défaut à 14 décimales des logarithmes en base 2 *)
  let lg2 = num_of_string "100000000000000"
  and lg3 = num_of_string "158496250072115"
  and lg5 = num_of_string "232192809488736"

  (* comparaison approchée puis exacte en cas de doute *)
  let (<<) (a,b,c) (x,y,z) =
    let a = a-x and b = b-y and c = c-z in
    let m = abs(b) + abs(c)
    and l = lg2*/Int(a) +/ lg3*/Int(b) +/ lg5*/Int(c) in
    if      l </ Int(-m) then true
    else if l >/ Int(m)  then false
    else begin
      Printf.fprintf stderr "précision insuffisante a=%d b=%d c=%d\n" a b c;
      Int(2)**/Int(a) */ Int(3)**/Int(b) */ Int(5)**/Int(c) </ Int(1)
    end

end (* Hnum *)


                     (* +---------------------+
                        |  Lance les calculs  |
                        +---------------------+ *)

(* conversion string -> num avec suffixe *)
let num_of_string(s) =
  let l = String.length(s) in
  let (e,i) = match s.[l-1] with
    |('0'..'9') -> 0, l
    | 'K' -> 3,  l-1
    | 'M' -> 6,  l-1
    | 'G' -> 9,  l-1
    | 'T' -> 12, l-1
    | 'P' -> 15, l-1
    | 'E' -> 18, l-1
    | 'Z' -> 21, l-1
    | 'Y' -> 24, l-1
    |_    -> failwith "suffixe inconnu"
  in Num.num_of_string(String.sub s 0 i) */ (Int 10)**/(Int e)

module H = Hamming(Hnum)
let _ =
  let k = num_of_string(Sys.argv.(1)) in
  let a = H.hamming(k) in
```

```
  print_endline(Hnum.string a)
```

5.8 Méthode volumétrique

```
(*+----------------------------------------------------------------------+
  |                                                                      |
  |                  Calcul du n-ème nombre de Hamming                   |
  |                                                                      |
  |                       Méthode volumétrique                           |
  |                                                                      |
  +----------------------------------------------------------------------+*)

(* Michel Quercia, le 08/06/99 *)
(* compilation :
   ocamlopt -o hamming nums.cmxa unix.cmxa volume.ml -cclib -lnums -cclib -lunix
*)

                         (*+-------------------------+
                           |  Messages de déboguage  |
                           +-------------------------+*)

let debug = ref false   (* option -debug *)
open Printf
open Unix

let t0 = ref(0.0) (* dernier relevé chrono *)

let message texte = if !debug then begin
  let t1 = (let t = times() in t.tms_utime +. t.tms_stime) in
  fprintf Pervasives.stderr "%5.2f %5.2f %s" t1 (t1 -. !t0) (texte());
  t0 := t1;
  flush Pervasives.stderr
end

let mesg texte = if !debug then begin
  prerr_string(texte());
  flush Pervasives.stderr
end



(*+----------------------------------------------------------------------+
  |                                                                      |
  |                        Descriptions abstraites                       |
  |                                                                      |
  +----------------------------------------------------------------------+*)

                           (* +-------------------+
                              |  nombres entiers  |
                              +-------------------+ *)

module type Entiers = sig
```

```
  type t

  val ( ++ ) : t -> t -> t                 (* opérations usuelles *)
  val ( -- ) : t -> t -> t
  val ( ** ) : t -> t -> t
  val ( // ) : t -> t -> t
  val (<<= ) : t -> t -> bool
  val ( == ) : t -> t -> bool

  val zéro   : t                (* on aura besoin de tous ceux là *)
  val un     : t
  val deux   : t
  val trois  : t
  val six    : t
  val douze  : t

  val of_string : string -> t           (* fonctions de conversion *)
  val string_of : t -> string

  val cubroot   : t -> t           (* racine cubique approximative *)
end

                          (* +---------------------+
                             |  Calcul de surface  |
                             +---------------------+ *)

module type Surf = functor (Z:Entiers) -> sig
  open Z
  type coef                                (* coefficients a,b     *)
  val  make    : t * t -> coef             (* enregistre a,b       *)
  val  surface : coef -> t -> t            (* surface ax+by <= n   *)

end

                          (* +--------------------+
                             |  Calcul de volume  |
                             +--------------------+ *)

module type Vol = functor (Z:Entiers) -> sig
  open Z
  type coef                          (* coefficients a,b,c, hmax et pmax *)
  val  make    : t * t * t * t * int -> coef    (* enregistre a,b,c,h,p  *)
  val  volume  : coef -> t -> t                 (* volume ax+by+cz <= n  *)
end

            (* +-------------------------------------------------+
               |  Recherche d'un entier de Hamming ou d'un rang  |
               +-------------------------------------------------+ *)

module type Ham = functor (Z:Entiers) -> sig
  open Z
  val hamming : t -> int -> t -> t -> t -> t -> unit
  val rang    : t -> int -> t -> t -> t -> t -> t -> t -> unit
```

```
  (* hamming   hmax  pmax   a    b    c    k          *)
  (* rang      hmax  pmax   a    b    c    x    y    z *)
end

(* +--------------------------------------------------------------------+
   |                                                                    |
   |                        Implémentation des entiers                  |
   |                                                                    |
   +--------------------------------------------------------------------+ *)

(* entiers de longueur arbitraire *)
module Znum = struct
  open Num
  type t = num
  let ( ++ )    = ( +/ )
  and ( -- )    = ( -/ )
  and ( ** )    = ( */ )
  and ( // )    = quo_num
  and (<<= )    = (<=/ )
  and ( == )    = ( =/ )
  and zéro      = Int(0)
  and un        = Int(1)
  and deux      = Int(2)
  and trois     = Int(3)
  and six       = Int(6)
  and douze     = Int(12)
  and of_string = num_of_string
  and string_of = string_of_num
  and cubroot x = let y = floor(float_of_num(x)**(1.0/.3.0)) in
                  num_of_string(Printf.sprintf "%0.0f" y)
end

(* flottants = entiers de 64 bits *)
module Zfloat = struct
  open Pervasives
  type t = float
  let ( ++ )    = ( +. )
  and ( -- )    = ( -. )
  and ( ** )    = ( *. )
  and ( // )    = fun x y -> floor(x/.y)
  and (<<= )    = (<= )
  and ( == )    = ( = )
  and zéro      = 0.
  and un        = 1.
  and deux      = 2.
  and trois     = 3.
  and six       = 6.
  and douze     = 12.
  and of_string = float_of_string
  and string_of x = Printf.sprintf "%0.0f" x
  and cubroot x = floor(x**(1.0/.3.0))
end

(*+----------------------------------------------------------------------+
```

```
  |                                                                      |
  |                       Surface d'un triangle                          |
  |                                                                      |
  +----------------------------------------------------------------------+*)

module Surface:Surf = functor(Z:Entiers) -> struct
  open Z

  (* a,b  = coefficients du triangle.              *)
  (* q    = b/a                                    *)
  (* next = triangle suivant s'il a été déjà visité *)
  type coef = {a:t; b:t; q:t; next:coef Lazy.t}


  (* enregistre a,b *)
  let rec make(u,v) =
    let (a,b) = if u <<= v then (u,v) else (v,u) in
    let rec c = {a=u; b=v; q=v//u; next=lazy(prolonge c)} in c

  and prolonge ({a=a; b=b; q=q} as coef) = let r = b -- q**a in make(r,a)

  (* calcul récursif de surface *)
  let rec surf {a=a; b=b; q=q; next=next} n s =
    if b <<= n then                             (* hauteur >= 1 *)
      let nb = n//b and qnb = (n**q)//b in
      surf (Lazy.force next)
           (n -- a**(qnb ++ un))
           (s ++ ((nb ++ un)**(deux**qnb -- q**nb ++ deux))//deux)
    else if zéro <<= n then s ++ (n // a ++ un) (* hauteur < 1 *)
    else s

  let surface coef n = surf coef n zéro

end (* Surface *)

(*+----------------------------------------------------------------------+
  |                                                                      |
  |                       Volume d'un tétraèdre                          |
  |                                                                      |
  +----------------------------------------------------------------------+*)

module Volume:Vol = functor (Z:Entiers) -> struct

  module S = Surface(Z)
  open Z

  (* a,b,c = coefficients du tétraèdre        *)
  (* p     = profondeur maximale              *)
  (* h     = hauteur additions brutales       *)
  (* next  = détail des calculs selon les cas *)
  type coef  = {a:t; b:t; c:t; h:t; p:int; next:calcul Lazy.t}
  and calcul =
    | Clos  of t                        (* formule close             *)
    | Sym   of coef * S.coef            (* a=b, 1/2-tétr. et tranche *)
```

```
  | Asym1 of t * coef * coef * S.coef   (* cas emboité               *)
  | Asym2 of t * coef * coef * S.coef   (* cas croisé                *)
        (* q    tét+   tét-   face(a,b) *)

(* enregistre les coefficients *)
let rec make(u,v,w,h,p) =
  let (u,v)   = if v<<=u then (v,u)   else (u,v) in
  let (a,b,c) = if v<<=w then (u,v,w) else if u<<=w then (u,w,v) else (w,u,v) in
  let rec coef = {a=a; b=b; c=c; h=h; p=p; next=lazy(prolonge coef)} in coef

and prolonge ({a=a; b=b; c=c; h=h; p=p} as coef) =
  if a == b then
    let q = c//a in if c == q**a
      then Clos(q)
      else Sym(make(a,deux**a,c,h,p), S.make(deux**a,c))

  else let q = (c++b--un)//b in if q**a <<= c
  then Asym2(q, make(a,b--a,c--q**a,h,p-1), make(b--a,b,q**b--c,h,p-1), S.make(a,b))

  else let q = q--un in
  Asym1(q, make(a,b--a,c--q**a,h,p-1), make(b--a,b,c--q**b,h,p-1), S.make(a,b))

(* formule close V(1,1,q,N) *)
let v(q,n) =
  let r   = n//q  in
  let num = r++un in
  let den = r**(deux**r++un)**q**q -- trois**r**q**(deux**n++trois)
                              ++ six**(n++un)**(n++deux)
  in (num ** den)//douze

(* addition des surfaces horizontales *)
let rec addition coef c n v =
  if zéro <<= n then addition coef c (n--c) (v ++ S.surface coef n)
  else v

(* calcul récursif du volume, addition de surfaces si profondeur = 0 ou hauteur <= hmax *)
let rec volume ({a=a; b=b; c=c; h=h; p=p; next=next} as coef) n =

  if b <<= n then match Lazy.force next with (* hauteur >= 1 *)
    | Clos(q)  -> v(q,n//a)
    | Sym(c,t) -> deux**(volume c (n--a)) ++ (S.surface t n)

    | Asym1(q,g,d,t) -> if h**c <<= n & p <> 0
      then let qnc = (q**n)//c in
           v(q,qnc) ++ (volume g (n -- a**qnc --a))
                    -- (volume d (n ++ a -- b**(qnc++deux)))
      else addition t c n zéro

    | Asym2(q,g,d,t) -> if h**c <<= n & p <> 0
      then let qnc = (q**n)//c in
           v(q,qnc) ++ (volume g (n -- a**qnc -- a))
                    -- (volume d (b**qnc -- n -- un))
      else addition t c n zéro
```

```
    else if zéro <<= n then (n//a)++un else zéro    (* h < 1 *)

end (* Volume *)


(*+-------------------------------------------------------------------------+
  |                                                                         |
  |               Recherche d'un entier de Hamming ou d'un rang             |
  |                                                                         |
  +-------------------------------------------------------------------------+*)

module Hamming:Ham = functor (Z:Entiers) -> struct

  module S = Surface(Z)
  module V = Volume(Z)
  open Z
  open S
  open V

  (* vérifie un volume *)
  let check coef n v =

    message(fun _ -> sprintf "contrôle %s -> " (string_of n));
    let k = volume coef n in
    mesg(fun _ -> sprintf "%s\n" (string_of k));
    if not(k==v) then failwith "contrôle négatif"


  (* tri selon l'ordre de Hamming *)
  let cmp a b c (x,y,z) (x',y',z') =
    let d = a**(x--x') ++ b**(y--y') ++ c**(z--z') in
         if d ++ y ++ z <<= zéro   then true
    else if zéro <<= d -- y' -- z' then false
    else failwith "comparaison douteuse"

  let nth a b c l n =
    let rec loop n l = if n == zéro then List.hd l else loop (n--un) (List.tl l) in
    loop n (Sort.list (cmp a b c) l)

                  (*+-------------------------------------+
                    |  Encadre le k-ème nombre de Hamming  |
                    +-------------------------------------+*)

  let recherche hmax pmax a b c k =

    let deuxabc = deux ** a ** b ** c  in
    let sixabc  = trois ** deuxabc     in
    let coef = V.make(a,b,c,hmax,pmax) in

    (* applique la méthode de Newton tant que l'encadrement se resserre *)
    (* hypothèse : nmin <= n < nmax, vmin <= k < vmax *)
    let rec newton(nmin,vmin,nmax,vmax,n,k) =
      if vmax -- vmin <<= douze then (nmin,vmin,nmax,vmax)
      else begin
```

```
    message(fun _ -> sprintf "Newton   %s -> " (string_of n));
    let v = volume coef n in
    mesg(fun _ -> sprintf "%s\n" (string_of v));
    if v == k then (n,v,nmax,vmax)
    else let n' = n -- (deuxabc**(v--k))//(n**n) in if v <<= k

      then if nmax <<= n' or v -- vmin <<= douze or vmax -- v <<= douze
      then linéaire(n,v,nmax,vmax,k)
      else newton(n,v,nmax,vmax,n',k)

      else if n' <<= nmin or v -- vmin <<= douze or vmax -- v <<= douze
      then linéaire(nmin,vmin,n,v,k)
      else newton(nmin,vmin,n,v,n',k)

  end

(* termine la recherche par interpolation linéaire *)
and linéaire(nmin,vmin,nmax,vmax,k) =
  if vmax -- vmin <<= douze then (nmin,vmin,nmax,vmax)
  else begin

    let n = nmax -- ((vmax--k)**(nmax--nmin))//(vmax--vmin) in
    message(fun _ -> sprintf "linéaire %s -> " (string_of n));
    let v = volume coef n in
    mesg(fun _ -> sprintf "%s\n" (string_of v));
         if v == k  then (n,v,nmax,vmax)
    else if v <<= k then linéaire(n,v,nmax,vmax,k)
    else                 linéaire(nmin,vmin,n,v,k)
  end
in

let n0 = cubroot(k**sixabc) in
let (p,u,q,v) = newton(zéro,zéro,n0++un,k++a++b++c,n0,k) in
if v -- u <<= douze then (p,u,q,v)
else let (_,_,q,v) = linéaire(p,u,q,v,k++deux) in (p,u,q,v)


     (* +--------------------------------------------------+
        |  Liste tous les points compris entre deux plans  |
        +--------------------------------------------------+ *)

let liste a b c p q d =

  message(fun _ -> "balayage ");
  let surf = surface (S.make(a,b)) in

  (* empile d points à y et z constants *)
  let rec ligne x y z d l =
    mesg(fun _ -> ".");
    if d == un then (x,y,z)::l else ligne (x--un) y z (d--un) ((x,y,z)::l) in

  (* empile d points à z constant, origine en (x,y) *)
  let rec triangle x y z p q d l =
```

```
    if not(b <<= q) then ligne (x++q//a) y z d l
    else
      let t = p//(deux**b) in
      let r = b**(t++un)   in
      let e = surf(q--r) -- surf(p--r) in
      let (d,l) = if e == zéro then (d,l)
                  else (d--e,triangle x (y++t++un) z (p--r) (q--r) e l) in
      if d == zéro then l
      else
        let t = p//(deux**a) in
        let r = a**(t++un)   in
        triangle (x++t++un) y z (p--r) (q--r) d l in

  (* empile d points à partir de la cote z *)
  let rec tétraèdre z p q d l =
    let e = surf(q) -- surf(p) in
    let (d,l) = if e == zéro then (d,l)
                else (d--e,triangle zéro zéro z p q e l) in
    if d == zéro then l
    else tétraèdre (z++un) (p--c) (q--c) d l in

  let l = tétraèdre zéro p q d [] in mesg(fun _ -> "\n"); l

              (*+-------------------------------------+
                |  Calcule le k-ème nombre de Hamming  |
                +-------------------------------------+*)

let hamming hmax pmax a b c k =

  let (p,u,q,v) = recherche hmax pmax a b c k in
  let coef = V.make(a,b++un,c++un,hmax,pmax)  in
  check coef p u;
  check coef q v;
  let l = liste a b c p q (v--u)   in
  let (x,y,z) = nth a b c l (k--u) in
  message(fun _ -> "terminé\n\n");
  printf "x=%s y=%s z=%s\n" (string_of x) (string_of y) (string_of z)


             (*+-----------------------------------------+
               |  Calcule le rang d'un nombre de Hamming  |
               +-----------------------------------------+*)

let rang hmax pmax a b c x y z =

  let n = x**a ++ y**b ++ z**c in
  let epsilon = n//b ++ un     in
  let nmin = n -- epsilon and nmax = n ++ epsilon in
  let coef  = V.make(a,b,c,hmax,pmax)
  and coef' = V.make(a,b++un,c++un,hmax,pmax) in

  (* calcule les volumes supérieur et inférieur *)
  let vol(n) =
    message(fun _ -> sprintf "plan sup %s -> " (string_of n));
```

```
      let k = volume coef n in
      mesg(fun _ -> sprintf "%s\n" (string_of k));
      check coef' n k;
      k
    in
    let rmin = vol(nmin) and rmax = vol(nmax) in
    message(fun _ -> "terminé\n\n");
    if rmax == rmin ++ un then printf "r=%s\n" (string_of rmin)
    else printf "%s <= r < %s\n" (string_of rmin) (string_of rmax)

end (* Hamming *)

(* +-----------------------------------------------------------------------+
   |                                                                       |
   |                        Interface ligne de commande                    |
   |                                                                       |
   +-----------------------------------------------------------------------+ *)

module Frontend(Z:Entiers) = struct

  module H = Hamming(Z)
  open Z
  open H
  open String


  (* Valeurs approchées de ln(2), ln(3), ln(5). Avec 50 chiffres on a de quoi voir venir *)
  let ln2 = "10000000000000000000000000000000000000000000000000"
  and ln3 = "15849625007211561814537389439478165087598144076924"
  and ln5 = "23219280948873623478703194294893901758648313930245"

  (* décode un nombre *)
  let int s =
    let l = length(s) in
    if l = 0 then zéro
    else let (e,i) = match s.[l-1] with
      |('0'..'9') -> 0, l
      | 'K' -> 3,  l-1
      | 'M' -> 6,  l-1
      | 'G' -> 9,  l-1
      | 'T' -> 12, l-1
      | 'P' -> 15, l-1
      | 'E' -> 18, l-1
      | 'Z' -> 21, l-1
      | 'Y' -> 24, l-1
      |_    -> failwith "suffixe inconnu"
    in of_string ((sub s 0 i) ^ (make e '0'))

  let main hmax pmax x y z =
  (* si z est défini on veut un rang. Sinon on cherche le y-ème nb   *)
  (* de Hamming. Bidouillage (infâme ?) pour avoir le -n facultatif. *)

    let rg = z <> "" in
    let hmax = int(hmax) and x = int(x) and y = int(y) and z = int(z) in
```

```
    let l = length((string_of x) ^ (string_of y) ^ (string_of z)) + 4 in
    let a = of_string(sub ln2 0 l)
    and b = of_string(sub ln3 0 l)
    and c = of_string(sub ln5 0 l) in

    if rg then rang    hmax pmax a b c x y z
          else hamming hmax pmax a b c y

end (* Frontend *)

(* lance les calculs *)
open Arg

let x     = ref "" and y = ref "" and z = ref ""
and float = ref false
and hmax  = ref "1000"
and pmax  = ref(5)

let helpmsg = "usage : " ^ Sys.argv.(0) ^ " (-r x y z | -n n) [-h hmax] [-p prof] [-debug] ( -float | -num)]
Par défaut : -n, -h 1000, -p 5.
"
let _ =
  parse [ ("-r",        String(fun s -> x := s),    "    cherche un rang");
          ("-n",        String(fun s -> y := s),    "    cherche un nombre");
          ("-h",        String(fun s -> hmax := s), "    hauteur seuil");
          ("-p",        Int   (fun p -> pmax := p), "    profondeur maxi");
          ("-debug",    Set(debug),                 "    débogage");
          ("-float",    Set(float),                 "    entier = flottant");
          ("-num",      Clear(float),               "    entier = num") ]
          (fun s -> if !y = "" then y := s else z := s)
          helpmsg

module Fnum   = Frontend(Znum)
module Ffloat = Frontend(Zfloat)

let _ =
  if !y <> ""
  then (if !float then Ffloat.main else Fnum.main) !hmax !pmax !x !y !z
  else prerr_endline helpmsg
```

Table des matières